VR三维技术系列

GLSL

渲染编程基础与实例

（C#版本）

赵　辉　楚含进　王晓玲　编著

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

VR 三维技术系列

GLSL 渲染编程基础与实例（C#版本）

赵　辉　楚含进　王晓玲　编著

電子工業出版社

Publishing House of Electronics Industry

北京·BEIJING

内容简介

本书介绍了用GLSL语言进行三维渲染的方法，以及用大量的实例来展示如何进行GLSL编程。本书详细讲述了GLSL渲染流程；GLSL着色器编程；顶点光照；像素光照；卡通渲染、影线渲染、分形渲染、Gooch渲染等非真实感渲染的实现；三维噪声的生成，以及噪声在云彩、木头纹理、大理石等渲染特效中的应用；棋盘、砖墙、Toyball等基于过程的渲染特效的实现；各种特殊光照效果渲染实现；通过GLSL进行图像处理的算法及实现。本书的特点是以各种渲染实例为核心，通过本书的学习，可以快速掌握GLSL语言的编程。

本书不仅可以作为数字媒体技术专业的专业基础课教材，还可以作为计算机学科和软件工程学科“数据结构和算法”、“计算机图形学”等课程的教材和参考书。

图书在版编目(CIP)数据

GLSL渲染编程基础与实例：C#版本/赵辉，楚含进，王晓玲编著. —北京：电子工业出版社，2017.7
(VR三维技术系列)
ISBN 978-7-121-31683-8

Ⅰ. ①G… Ⅱ. ①赵… ②楚… ③王… Ⅲ. ①三维动画软件－程序设计 Ⅳ. ①TP311.5

中国版本图书馆CIP数据核字（2017）第120543号

策划编辑：张 迪
责任编辑：底 波
印 刷：中国电影出版社印刷厂
装 订：三河市良远印务有限公司
出版发行：电子工业出版社
北京市海淀区万寿路173信箱 邮编100036
开 本：787×1092 1/16 印张：11.5 字数：294千字
版 次：2017年7月第1版
印 次：2017年7月第1次印刷
定 价：59.00元

所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：(010) 88254888，88258888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

本书咨询联系方式：(010)88254469，zhangdi@phei.com.cn。

序

2015 年以来，虚拟现实技术的应用在国际国内发展很快。教育、医疗、娱乐、影视、游戏、安全、交通等各行各业都对虚拟现实技术进行了大量应用。虚拟现实技术的基础和核心是三维计算机图形学，分为四大模块：建模、渲染、动画、交互。目前国内大量的虚拟现实应用都局限于在西方开发的虚拟现实引擎的技术上进行开发的上层应用。我们这套丛书着重底层核心技术的讲解，三维计算机图形学在知识结构上来说需要数学、物理、工程、计算机编程、艺术五个方面。设计建模、渲染等算法需要微分几何、线性代数、概率统计等数学知识的理解和掌握；动画模拟需要流体、刚体等物理知识的理解和掌握；把这些数学、物理理论变为程序需要极强的编码能力，也就是从理论到实践的工程能力；三维图形学的最终表现形式是视觉上可看得到的，因此也需要良好的艺术修养和审美。虚拟现实和它所依赖的三维计算机图形学特别适合锻炼并能够融会贯通学生的数学、物理、工程、编程和艺术能力。三维计算机图形学是一个跨学科的领域，三维图形学处理的是三维模型数据，学生在这个领域中学到的数学建模、工程等能力，也可以用到其他行业，如人工智能等，对其他行业的大数据进行分析和处理。

2008 年以来，全国各个高等院校纷纷在各自软件工程学科专业的基础上开设了数字媒体技术专业。数字媒体技术专业和计算机科学专业的区别是，前者主要是着重学习二维图像和三维图形相关的算法和应用开发，而后者还需要学习其他计算机科学相关的知识。由于开设和建立时间短，各学校的数字媒体技术专业的教学工作都还处在摸索阶段，也没有形成统一、成熟的教材体系。根据在数字媒体技术专业多年的教学实践经验，我们总结出本专业要以计算机三维图形学的理论和算法为基础，以三维应用开发为导向进行建设。

根据多年一线教学经验与反馈，以及当前的三维图形学研究成果，我们编写了本套丛书。本套丛书涵盖了三维图形学算法的三个方面：建模、动画和渲染。内容根据数字媒体技术专业的教学特点分散到 5 本 VR 三维技术系列图书中。通过本系列专业图书，再加上已有的成熟的计算机基础编程教材，以及三维软件使用的教材，就可以完整地覆盖数字媒体技术专业的所有课程。

书里的代码采用 C#编程语言。C#编程语言是一种结合了 C ++ 和 Java 优点的编程语言。C#语言相对于其他编程语言来说比较容易学习和掌握，但是本套丛书里讲述的原理和算法不仅限于 C#语言。读者可以通过示例中的代码，采用自己熟悉的编程语言来进行编程。本套丛书包含了很多计算机图形学会议 Siggraph 论文里最新的、核心的、关键突破和进展的图形学算法讲解、实现和分析。

前　言

虚拟现实应用离不开逼真的渲染。渲染分为实时渲染和非实时渲染两大类。在影视特效中，常用到的是非实时渲染。而在游戏等应用中，需要能够实时显示的渲染技术。非实时的渲染通过对光照进行物理模拟，从而达到和相机拍摄无法区分的效果。但由于要进行光线追踪等计算，这种算法耗时很长。而实时渲染由于对物理光照进行了大量的简化，从而可以很快速地进行计算。

在 GPU 上进行实时渲染，是目前成熟的解决方案。需要用特定的编程语言来对 GPU 进行编程，从而使 GPU 能执行设计好的光照公式。对于不同的光照程序，可以得到不同的渲染效果。如果只用 OpenGL 进行渲染，那么就受限于 OpenGL 内置的渲染效果。GPU 渲染是三维游戏、虚拟现实场景等应用中一个最重要的核心模块。本书提供了 GPU 编程的基础、代码和实例。

本书介绍了用 GLSL 语言进行三维渲染的方法，以及用大量的实例来展示如何进行 GLSL 编程。本书详细讲述了 GLSL 渲染流程；GLSL 着色器编程；顶点光照；像素光照；卡通渲染、影线渲染、分形渲染、Gooch 渲染等非真实感渲染的实现；三维噪声的生成，以及噪声在云彩、木头纹理、大理石等渲染特效中的应用；棋盘、砖墙、Toyball 等基于过程的渲染特效的实现；各种特殊光照效果渲染的实现；通过 GLSL 进行图像处理的算法及实现。本书的特点是以各种渲染实例为核心，通过学习本书的内容，可以快速掌握 GLSL 语言的编程。

本书不仅可以作为数字媒体技术专业的专业基础课，还可以作为计算机学科和软件工程学科“数据结构和算法”、“计算机图形学”等课程的教材和参考书。需要书中部分代码的读者，可发邮件向作者索取，邮箱地址：graphicsresearch@ qq. com。

赵　辉

2017 年 5 月于美国哈佛大学

作者简介

赵辉，虚拟现实专家、清华大学丘成桐数学科学中心访问学者、哈佛大学访问学者。主要研究计算微分几何、拓扑、三维模型处理算法（三维模型简化、细分、分割、变形、光滑、参数化、向量场、四边形化等）、三维动画算法（骨骼动画、蒙皮算法）、渲染算法（非真实感渲染、实时渲染、基于物理渲染），以及三维技术在3D打印、虚拟现实、增强现实、三维游戏、手机游戏、影视特效等的应用。

楚含进，现任AMD中国区VR与计算平台总监，负责图形处理器（GPU）技术在虚拟现实（VR）中的应用，以及游戏设计、计算机图形与仿真等技术领域的应用和合作，是国内VR产业早期从业者，为国内VR媒体撰写各类有关文章。同时，在异构计算领域，推动将GPU异构计算用于机器学习、计算机视觉领域。曾带领团队先后将GPU异构计算贡献于OpenCV，以及Caffee、MLP等开源项目，主导引进并支持多个有关OpenCL计算的书籍。

王晓玲，北京科技大学教授，美国西北大学、哈佛大学访问学者，有限元模拟、机械仿真，物体相变、生物材料分析、三维打印材料专家。

目　录

第1章 GPU与图形应用编程介绍

1.1 GPU发展史与Shader

提到Shader编程就必须讲述GPU的历史，尤其伴随着当代虚拟现实（VR）技术的兴起，GPU作为图形处理器更是VR系统的核心，对GPU的了解是编写VR程序的必备之路。谈及GPU的发展史，每个读者都有自己的坐标，网上也有GPU的历史和各厂家的产品介绍等相关信息。今天我们从GPU与Shader可编程的角度来了解GPU的发展。

让我们把目光放在GPU产生之后的日子里。早期的显卡只是为显示输出而设计的专用加速器，而GPU的概念最早在1999年由NVidia提出，随着显卡芯片的技术变革和飞速发展，其用途和概念已经超出了功能单一的图形硬件加速器，渐渐向使用灵活功能强大的通用计算处理器进化。这一举动标志着GPU在PC主机里地位的不断上升。

OpenGL是出现于20世纪90年代初的专业图像API，并很快成为了个人计算机领域图像发展的主导力量和硬件发展的动力。虽然OpenGL的影响带起了广泛的硬件支持，但在当时用软件实现的OpenGL仍然很普遍。20世纪90年代末，DirectX开始受到Windows游戏开发商的欢迎。不同于OpenGL，微软坚持提供严格的一对一硬件支持。这种做法使得DirectX身为单一的图形API方案并不得人心，因为许多的GPU也提供自己独特的功能，而当时的OpenGL应用程序已经能满足它们，导致DirectX往往落后于OpenGL一代。随着时间的推移，微软开始与硬件开发商展开更紧密的合作，并开始重视DirectX的发布与图形硬件的支持。

Direct3D 5.0是第一个增长迅速的API版本，而且在游戏市场中迅速获得普及，并直接与一些专有图形库竞争，而OpenGL仍保持重要的地位。Direct3D 7.0支持硬件加速坐标转换和光源（T&L）。此时，3D加速器由原本只是简单的栅格器发展到另一个重要的阶段，并加入3D渲染流水线。硬件坐标转换和光源（两者已经是OpenGL拥有的功能）于20世纪90年代在硬件中出现，为往后更为灵活和可编程的像素着色引擎和顶点着色引擎设置了先例。2000年后，随着OpenGL API和DirectX类似功能的出现，GPU增加了可编程着色的功能。现在，每个像素都可以经由独立的小程序处理，当中可以包含额外的图像纹理输入，而每个几何顶点同样可以在投影到屏幕上之前被独立的小程序处理。如果说GPU名词的出现体现出业界对显卡发展的方向和态度，那么ATI（AMD Radeon显卡所在公司的前身）在2002年发布的首款支持具有划时代意义的DirectX 9图形API的R300系列GPU（即大名鼎鼎的Radeon 9700和9500系列）从根本上将GPU推上了前进的轨道。它是世界上首个Direct3D 9.0加速器，像素和顶点着色引擎可以运行循环和长时间的浮点运算，就如CPU般灵活，并达到了更快的图像数组运算。像素着色通常被用于凹凸纹理映射，使对象通过增加

纹理呈现更加丰富的明亮、阴暗、粗糙，或是偏圆及被挤压效果。虽然 DirectX 8 率先提出了 GPU 可编程概念，从而引出了最早的 Vertex Shader 与 Pixel Shader，但真正使两者发扬光大并大规模应用的是 DirectX 9。也是从这时候开始，微软开始逆袭居统治地位长达 10 年之久的 OpenGL，由此可见，可编程 GPU 在 GPU 发展史上的重要意义，自此 3D 图形应用发展到了一个崭新的时代，而 DirectX 9 的统治时间更是长达 7 年之久，ATI 之后发布的 R400 和 R500 系列一直将优势保持到了 2006 年。在这一年，微软随着颇受争议的 Windows Vista 发布了 DirectX 10，由于这个系统推出得过于仓促及不合时宜，Windows Vista 和 DirectX 10 都没有得到广泛的应用。但 ATI 还是率先发布了支持 DirectX 10 的 R600 系列，这个版本主要为开发者带来了强大的 Geometry Shader（几何着色）及一些先进新特性，为 3D 图形编程又带来了新的元素。但随后 NVidia 同样发布了著名的支持 DirectX 10 的 GTX 8800 系列，暂时夺回了 GPU 性能宝座。正如前面提到的，DirectX 10 并没能取得成功，只是一个过渡产品。

不久，随着微软在 2009 年发布了 Windows 7，与之捆绑的 DirectX 11 走入了人们的视线，它也成为 DirectX 9 的终结者。DirectX 11 为 GPU 编程带来了众多新特性，Tessellation Shader、Compute Shader 及多线程支持。此时的 ATI 已被 AMD 收购，同年 AMD 又一次率先发布了支持 DirectX 11 的 Radeon HD 5000 系列，AMD 再次利用技术上的创新和优势引领 GPU 的发展，在 Radeon HD 5000 的整个生命周期里都没有遇到任何挑战。2010 年 AMD 又发布了 Radeon HD 6000 系列，继续巩固着胜果。就在此时，一个小插曲开始出现，故事源于 DirectX 11 里一个由 AMD 提出的全新技术 Tessellation。从广义上讲，Tessellation Shader 的设计与 DirectX 10 引入的 Geometry Shader 有相似之处，它们的本质都是在已有模型的基础上生成更多的图元或三角形，从而达到特定的效果。这两者为当代 GPU 编程流水线增色添彩，为开发者提供了更多的选择。但由于要实现这一功能开销很大，需要在 GPU 的设计中加入大量的硬件处理单元来生成三角形。而 Tessellation 的用途却比较有限，一般应用在使用 Displacement mapping 来生成地形等应用中。从前面的介绍可以看出，Tessellation 是一项功能单一却开销很大的技术，这也是为何要配备大量硬件处理单元进行计算的原因（道理等同于硬件编解码）。这一技术后来被 AMD 的对手 NVidia 利用，着力加强对 Tessellation 的利用，在 GPU 设计中加入大量的三角形生成与处理单元，这样就可以在处理图形渲染过程中最基本的元素三角形时获得巨大优势，这一点在游戏巫师 3 和古墓 10 中体现尤其明显，两者都使用了大量的 Tessellation。这一点与 AMD 显卡的技术路线完全不同，此时 AMD 已经开始研究为下一代 DirectX 12 带来事实标准的 Mantle。无奈 NVidia 的市场和跟随策略非常成功，而 AMD 内部动荡，这一时期一直是 NVidia 占据着市场的主导位置。尽管如此，AMD 随后推出的 Mantle 标准技术也非常成功，促使了 DirectX 12，OpenGL 的下一代 Vulkan 甚至苹果公司的 Metal 图形接口标准的出现。依旧坚定地推动 GPU 的发展，为 GPU 不断注入新鲜的活力。伴随着 Mantle，2012 年，AMD 推出了新一代架构 GCN，旨在提升 GPU 的通用计算性能，这一理念与 DirectX 11 设计一脉相承，引入了 ACE（Asynchronous Compute Engine），要知道这就是后来带动了 DirectX 和 OpenGL 变革的核心技术，也是 DirectX 12 和 Vulkan 为开发者带来的最大甜品。无独有偶，2014 年 Oculus 被 Facebook 收购，虚拟现实一夜之间成为下一代互联网计算平台，但是虚拟现实所需要的图形处理能力需要指数级的增长，AMD GPU 的 ACE 架构将 VR 对图形处理能力的需求在不损失质量的情况下降低了一个级别，这也就是为什么 Oculus 的 VR 头显对 AMD 的显卡支持明显要求低的原因。

Compute Shader是DirectX 11的最重量级特性，它的灵活性和强大功能远超GPU渲染流水线中的其他成员，它的引入使得以往不可能的特效变为了现实，通过物理计算来模拟现实世界中的各种效果，比如毛发渲染、布料渲染、草地渲染、光线追踪、物理碰撞等。从技术上来说，Compute Shader、异步处理单元等都代表着未来的图形技术发展方向，未来GPU将越来越强调通用计算性能，GPU越来越像CPU。无奈过去几年AMD在软件、工具等方面并没有提供足够的用户体验给开发者，以至于流失很多开发者转向NVidia的显卡，但不能否认AMD的显卡架构师对未来图形处理的远见卓识。经过几年调整，AMD在CPU上刚刚发布AMD锐龙Ryzen系列，10年后回归CPU主场。同时，在2017年，AMD还将发布全新Vega架构系列GPU，事隔5年后，AMD再一次大刀阔斧地对GPU架构进行变革，引入NCU（Next-generation Compute Unit），并对流水线进行大幅改动。未来GPU发展趋势如何，让我们拭目以待。

1.2 GLSL Shader编程在图形设计中的作用

首先了解GPU进入可编程时代的发展史。

Shader其实就是一段执行在GPU上的程序，此程序使用OpenGL ES SL语言来编写。最早的GPU仅仅由几种不同的硬件处理单元组成，进行最基本的3D图形加速功能。从2003年发布的DirectX 8.0和OpenGL 1.5开始引入了可编程特性。最早被提出的GPU渲染中最核心的两个组件是Vertex Shader和Fragment Shader。2004年发布的OpenGL 2.0则伴随着全新的GLSL标准为GPU编程带来了划时代的意义。GLSL由当时的GPU巨头3DLabs提出并完善。当时的GPU为开发者提供了两种不同类型的可编程处理器：Vertex Processor和Fragment Processor，基于这两种处理器，开发者有能力开发出比Fix-Function流水线更加丰富的特效。

此后，OpenGL陷入了长达6年的停滞期。在此期间微软发布了DirectX 9和DirectX 10，引入HLSL和一系列高级特性，将OpenGL甩在身后。大量游戏转投微软阵营，一时间基于OpenGL的游戏除了Doom和Quake系列几乎绝迹。而DirectX 10更是将一直分离的Vertex Processor和Fragment Processor进行统一，ATI于2006年提出了Unified Shader Model概念，不同的Shader编程全部由Streaming Processor来进行计算，同时又提出了一个全新的Geometry Shader，这也是GPU通用计算的雏形。随后的OpenGL 3.0只是在前者上修修补补，没有太多新意。

直至2010年OpenGL 4.x的发布，OpenGL才在特性上追上了微软的脚步，同时微软提出了大名鼎鼎的DirectX 11，直到今天它还是被应用最广泛的桌面图形API，它的统治时间更是长达8年之久。随着DirectX 11的出现，GPU架构的硬件特性又达到了一个新高度，由AMD提出的Tessellation Shader和Compute Shader以及CPU多线程渲染被引入。OpenGL虽然也在4.0版和4.3版分别加入了Tessellation Shader和Compute Shader的支持，但至今仍不支持CPU多线程渲染特性。

这一情境在全新图形API Vulkan面世后被终结，微软和Khronos组织分别于2015年和2016年发布了全新的DirectX 12和Vulkan，这两种API都秉承了AMD Mantle的设计理念，大幅降低了驱动开销，支持多线程渲染，并支持由AMD提出的Asynchronous Compute Engine异步计算引擎。目前GLSL的最新版本为4.5，Vulkan版本的GLSL的标准也正在制定中，相信不久的将来便会有完整的文档出现（目前开发者可参考OpenGL版的GLSL来进行Sha-

der 编程，它们的绝大多数用法是一致的）。

接下来，我们来看看 Shader 编程在图形设计中的作用。

通过前面的介绍我们可以看出，Shader 编程在图形设计中的重要性越来越高，它决定了图形效果的丰富程度和性能表现。

简单来说，图形程序的设计理念是让 GPU 承担尽可能多的计算任务，从而将 CPU 解放出来，之所以这样做有以下几个原因。首先 GPU 的架构设计与图形计算中的任务高度并行相吻合，而从硬件设计角度讲，扩充 GPU 的宽度（流处理器个数）要远比扩充 GPU 的高度（流处理器频率）容易，道理与 CPU 向着多核方向发展相同，都是在无法大幅提升主频的情况下通过增加宽度来提高处理器的性能。其次 GPU 在 PC 中的地位除了进行图形运算外还负责图像输出，也就是我们从显示器上看到的一切都是经由 GPU 处理完成并呈现的。我们知道 CPU 与 GPU 享有着不同的存储空间，CPU 的存储空间就是我们常见的内存，而 GPU 的存储空间则是其自身配备的调整显存。在计算机的硬件体系中，两者间的数据交互需要通过 PCIE 总线，虽然这个高级的总线在 PC 中的传输速度出类拔萃，但相比内存与显存来说，这显然是瓶颈。因此，在程序的设计中要尽可能避免或减少内存与显存之间的数据传输。如今的图形计算流程是由 CPU 准备数据的，这些数据可能存储在速度相对较慢的外部存储器中，如固态硬盘或磁盘，之后再将数据从内存传递到显存中，由 GPU 进行计算处理最终输出到显示终端。通过分析这个处理流程我们可以看出，在整个过程中涉及一次数据从内存到显存的传输，这是我们能够做到的最优方案，虽然这个时间也会比较长，但我们可以通过游戏启动时的加载过程或通过多线程预加载来完成，也就是说这个时间不会影响到游戏运行时的处理速度。这也就解释了为什么我们希望 GPU 来尽可能多地完成计算处理，这样就不需要在游戏中将 GPU 的数据从显存传回内存，待 CPU 处理完成后再返回到显存的漫长过程了。

看到这里我们应该明白 Shader 编程在图形设计中的作用了，我们希望能够使 GPU 形成一个单独的自治区域，所有的任务都高度自治，GPU 的重要性不言而喻。AMD 的图形技术发展也在逐步印证这一点，从最初的 Vertex Shader 和 Fragment Shader（Pixel Shader），到后来逐步完善的 Geometry Shader、Tessellation Control Shader（Hull Shader）和 Tessellation Evaluation Shader（Domain Shader），以及现在的 Compute Shader。这一系列特性的加入都是围绕着使 GPU 成为独立计算单元的目标进行的。

下面让我们来一一介绍这些 Shader 的作用及地位。

Vertex Shader 是最常见的历史最悠久的 Shader，它主要用于将 3D 模型的顶点坐标从 3D 虚拟空间转变成 2D 屏幕坐标和深度缓冲。Vertex Shader 是图形流水线的第一站，也是每个流水线必须包含的阶段。Vertex Shader 是必然存在的。Vertex shader 主要完成以下工作：基于点操作的矩阵乘法位置变换；根据光照公式计算每点的 color 值；生成或者转换纹理坐标。

Tessellation Shaders 是在 OpenGL 4.0 和 DirectX 11 中引入的，它们包含了两个阶段，分别是 Tessellation Control Shader 及 Tessellation Evaluation Shader。它们可以在运行时生成大量的细分三角形来使一个粗粒度的三角形模型变成更细粒度的三角形模型。一般的使用场景包括根据距离远近使用 LoD（Level of Details）优化性能、通过 Displacement Mapping 和 Height Map 生成地形等。这两个 Shader 的位置处在 Vertex Shader 之后和 Geometry Shader 之前（如果图形流水线配备了 Geometry Shader，否则会被直接送入 Rasterizatoin 阶段，随后进入 Fragment Shader。Tessellation Shader 是可选的。

Geometry Shader是在OpenGL 3.2和DirectX 10中引入的，在这个阶段，开发者可以控制在Vertex Shader输入的3D模型基础上生成一些新的图形元素，包括点、线段、三角形等。在某些简单的图元生成上作用与Tessellation Shaders类似。同时Geometry Shader还提供了顶点流处理的能力，可以通过这个阶段为顶点缓冲中的顶点进行索引并输出到一个特定的缓冲区。在Geometry Shader里，我们处理的单元是Primative。虽然根本上都是顶点的处理，但进入Vertex Shader的是一次一个的顶点，而进入Geometry Shader的是一次一批的顶点，Geometry Shader掌握着这些顶点所组成的图元信息。Geometry Shader的处理阶段处于流水线的栅格化之前，也在视锥体裁剪和裁剪空间坐标归一化之前。虽说裁剪过程会剔除部分图元也会分割某些图元，但就目前来说，不会有其他流水线的可编程阶段会在Geometry Shader之后提供影响图元的性质（形式和数量）——这是Geometry Shader鉴于其位置的特殊性而拥有的一个重要特点。Geometry Shader程序在Vertex Shader程序执行之后执行。

Fragment Shader是除Vertex Shader之外另一个最常见的Shader，它与Vertex Shader一同出现，处于Rasterizatoin阶段之后，是整个图形流水线中唯一的2D Shader。Pixel Shader（像素着色器）就是众所周知的Fragment Shader（片元着色器），它计算每个像素的颜色和其他属性。通过应用光照值、凹凸贴图，阴影，镜面高光，半透明等处理来计算像素的颜色并输出。它也可改变像素的深度（z-buffering）或在多个渲染目标被激活的状态下输出多种颜色。一个Pixel Shader不能产生复杂的效果，因为它只在一个像素上进行操作，而不知道场景的几何形状。它的处理对象是在屏幕的坐标上输出颜色值，结果将被呈现在显示器上。Fragment Shader在整个图形流水线的Shader中是限制最少的，也是功能最强大的，一般较为复杂的计算将在这一阶段进行，来达到丰富输出效果的目的。

整个图形流水线的Shader就如同工厂流水线上的工人一样，每个不同的工种都被赋予了特定的权限和功能，它们在各自的环境下施展着不同的能力，因此它们的能力都是受限的。接下来我们来了解Shader中最具革命性的成员Compute Shader。

Compute Shader是在OpenGL 4.3和DirectX 11中引入的。它独立于图形流水线，可以存在于图形队列和计算队列的不同位置。这也是GPU Shader编程诞生以来出现的限制最少、功能最强大的Shader。它的能力几乎等同于GPGPU，也就是常说的OpenCL。在Compute Shader中，开发者被赋予最大的权限来访问GPU中的各类资源，使用Compute Shader可以编写出灵活性极高、功能极为强大的算法，来辅助图形编程中的效果、物理计算、模拟等高级特性。而AMD的TressFX（毛发渲染技术）就是一个非常成功的案例，TressFX的算法设计就是通过Compute Shader对毛发运动效果的模拟来达到实时毛发渲染的目的。而物理模拟计算也将成为未来提升图形渲染水平的重要趋势，如布料、草、树叶及光线追踪等效果。在GPU飞速发展的今天，将会有越来越多的复杂物理效果被应用于实时图形应用中。AMD引入Computer Shader的计算单元还有一个目的是最大限度保证GPU的性能发挥。现在市面上有很多硬件独立单元来做诸如物理开发，这在一段时期内是必然的，随着GPU的发展必然会过渡到Computer Shader。

笔者对AMD的显卡较为熟悉，所以下面以AMD显卡为例为大家说明一些问题。通过前面的介绍，我们了解了GPU图形渲染的每个重要环节，以及Compute Shader所扮演的重要角色。而这一切也反映在AMD GPU设计的理念中。目前已经发布的GPU中，从第一代GCN开始，代号为Tahiti、Hawaii、Fiji的旗舰产品在GPGPU通用计算中的性能都处在绝对

的领先地位，而将在 2017 年发布的 Vega 10 架构会将通用计算的性能再推向下一个巅峰。

接下来我们就看看 GCN 架构为通用计算带来了哪些特性。

如图 1-1 所示为 GPU 架构进化的过程，从最早的 Fixed Function 设计到 Radeon 9700 时代的简单 Shader 模型，再到 Radeon HD 5000 6000 系列的 VLIW5 和 VLIW4。最新的 GCN 架构相对 VLIW4 架构进行了彻底的颠覆革新。从图中我们可以看出 VLIW4 是一个超长指令架构，4 个红色矩形代表了在每次 ALU 计算中包含 4 条不相关指令。这样的设计非常适合早期的纯图形 Shader 计算，Fragment Shader 中的每个像素都完全独立，没有依赖关系，这样能够维持极高的 VLIW4 指令利用率，然而通用计算中由于分支很多，依赖关系复杂，这样的架构设计将导致大规模的资源浪费。

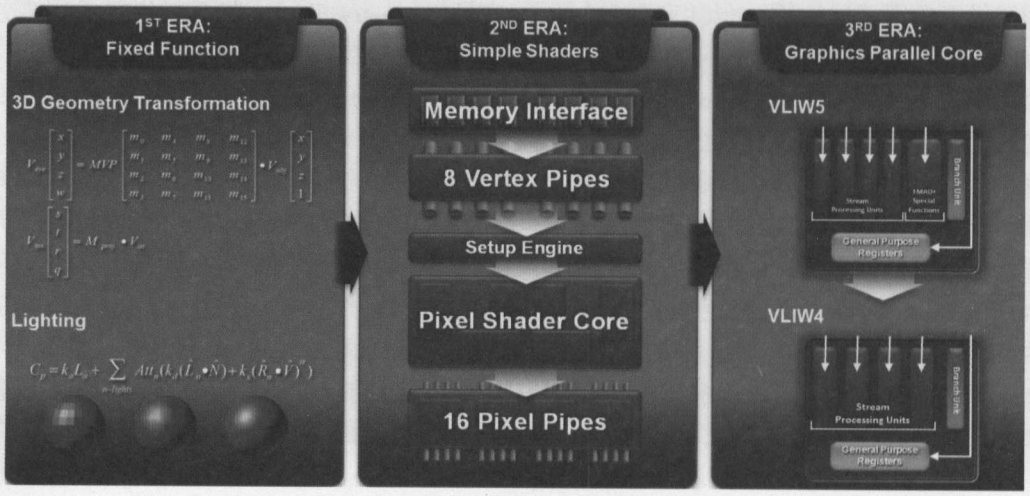


图 1-1　GPU 架构进化的过程

如图 1-2 所示很好地说明了这个问题，在 VLIW4 中，每个 CU 配备了 16 个 ALU、64 个流处理器及 1 个 SIMD 指令，每个 ALU 在每个周期可以对 4 个通道进行 4 次不同指令的计算，也正好对应着图形计算中的 RGBA 通道，因此非常适合图形计算。同时这样的架构非常依赖编译器来优化遍历每个周期所执行的指令尽可能达到 4 个。

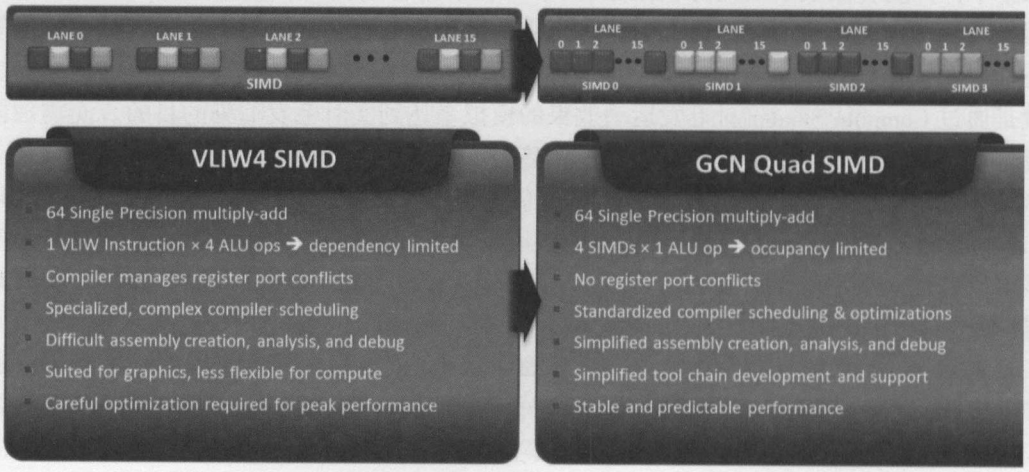


图 1-2　VLIW4 和 GCN 架构

而GCN架构则完全推翻了之前的设计来为通用计算保驾护航。可以看出，每个CU配备了4个ALU，可分别处理4个不同的SIMD指令，同样拥有64个流处理器。每个SIMD应用于16个流处理器，并且不再需要打包4个指令后再启动ALU，这就给指令的发射带来了很大的灵活性，同时也不需要编译器做特殊的优化，全部是比较规整的细粒度操作。

GCN架构还使每个CU里的64个流处理器同步执行，从而在使用LDS（Local Data Share）时不需要进行任何同步操作，去除了同步操作的开销。

另外，如图1-3所示，GCN架构的GPU还在每个CU里专门配备了Integer Atomic Units来高效地实现LDS原子操作的性能，Reduction操作在通用计算中非常常见，而在很多场合中Reduction操作都需要原子操作来实现。

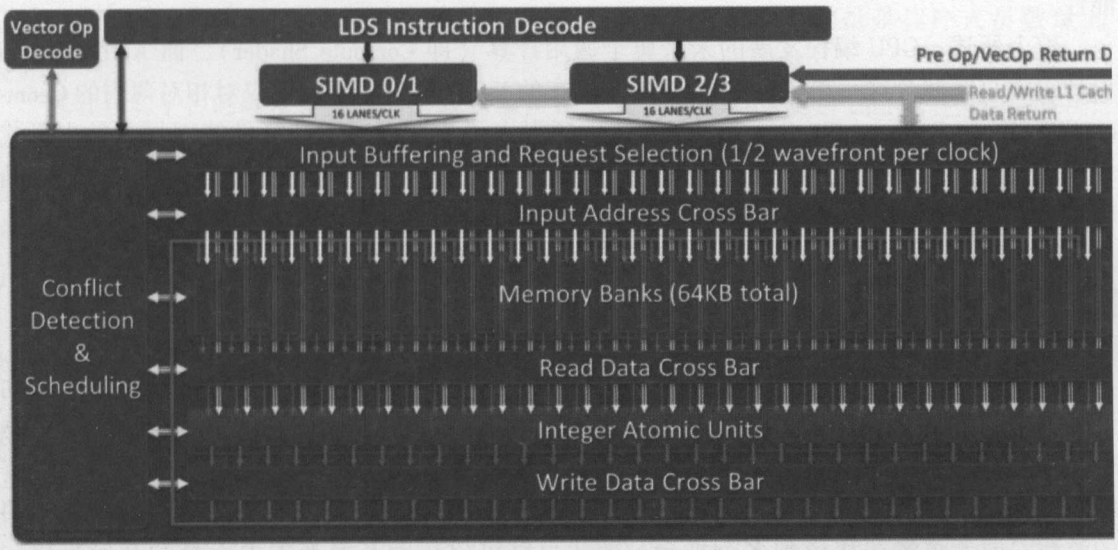


图1-3 Integer Atomic Units

除此之外，GCN架构的GPU还专门为CPU开辟了一块最大256MB的显存空间，CPU可以高效地对这片存储空间进行写操作，这样做的好处是，CPU可以高效地直接更新显存上的数据，同时保证GPU能以最高效的内存访问方式（即访问显存）来读取数据，从而节省一次内存到显存的复制。GPU访问内存的效率通常不高，远远低于访问显存的性能，而这往往成为GPU计算的最大瓶颈（GPU在进行通用计算中的瓶颈往往都来源于对显存和内存的访问）。

以上这些架构的重大变革被证实在通用计算中展示出巨大的实力，一经推出便所向披靡，横扫竞争对手，几年前兴起的挖矿热潮便是源于GPU的通用计算。直至今日，GCN架构的通用计算能力仍无法被超越。

最后，我们来说一说GCN架构为GPU图形和计算带来的最重磅特性，ACE（Asynchronous Compute Engine）异步计算引擎。这项由AMD设计并实现的硬件引擎成为了DirectX 12和Vulkan引入的最大特性，并且在它出现5年之后，仍没有竞争对手能够实现。ACE的设计思路是为了在图形渲染中能够更充分地利用流处理器等通用计算单元，由于流处理器只有在Shader执行的过程中才会被使用，而图形渲染过程中有大量的计算时间消耗在GPU的Fixed Function硬件处理单元上，如Vertex Assembler、Input Assembler、Tessellator

和 Rasterization 等。2016 年，已经有众多顶级游戏大作支持了 DirectX 12 和 Vulkan，而 DirectX 12 的被接纳速度也成为有史以来之最。AMD GPU 在众多 DirectX 12 和 Vulkan 游戏中表现抢眼，在最新发布的狙击精英 4 中，AMD GPU 在 DirectX 12 模式下借助 ACE 特性取得了 20% ~30% 的性能提升。

Compute Shader 不仅拥有对资源的最大访问权限，如 LDS、原子操作、AMD 专用的 ShaderInstrinsic Function 以及 Work Item、Work Group、Local ID、Global ID 等索引信息，而且还提供了极大的资源访问灵活性，开发者可以通过参数设置来决定每个流处理器与计算任务的映射，控制 Workgroup 的大小和维度，从而控制 LDS 的使用量、Wavefront 的数量。这里每一个细节都对通用计算中 GPU 资源的利用率起到关键作用，这也给性能优化带来最大的可能性。

综上所述，GPU 编程发展的未来属于通用计算（即 Compute Shader），而 AMD 的 GPU 设计也一直在迎合这一趋势，将于 2017 年发布的 Vega 10 GPU 架构不仅对相对薄弱的 Geometry Pipeline、Deferred Shading 等进行增强，还引入了为增强通用计算性能而提出的次世代计算单元 NCU（Next - generation Compute Unit），能够以双倍于单精度浮点性能来执行半精度浮点计算。AMD 将一如既往地为推动 GPU 技术的进化与发展不懈努力。

1.3 游戏引擎的发展

游戏引擎是为视频游戏的开发而设计的软件框架。开发人员使用它们来开发主机、移动设备和电脑游戏。游戏引擎提供的核心功能通常包括用于 2D 或 3D 图形的渲染引擎（“渲染器”）、物理引擎或碰撞检测（和碰撞响应）、声音、脚本、动画、人工智能、网络组件、资源和内存管理、多线程、场景编辑器。在很大程度上，通过使用相同的游戏引擎来创建不同的游戏，或者将游戏移植到多个平台，游戏引擎可以帮助开发者省去大量的开发流程和时间。

在许多情况下，除了可重用的软件组件之外，游戏引擎还提供了一套可视化开发工具。这些工具通常以数据驱动的方式提供简单、快速的游戏开发方式。引擎开发者通过开发几乎所有游戏构建需要的元素来组成强大的软件套件来帮助真正的游戏开发者摆脱这种重复的、高投入的、底层的工作，从而可以直接去创造游戏内容。

大多数游戏引擎套件提供了便于开发的组件，如图形、声音、物理和 AI 功能。这些游戏引擎有时被称为“中间件”，因为与中间件的商业意义一样，它们提供灵活和可重复使用的软件平台。其开箱即用的所有核心功能，成为开发游戏，同时降低成本和复杂性的关键因素。Unity、Unreal、Frostbite、Cocos2D/3D、CryEngine 便是这种被广泛使用的游戏引擎。

与其他中间件解决方案一样，游戏引擎通常提供跨平台功能，允许在各种平台上运行相同的游戏，包括游戏机和 PC，同时不用对游戏源代码做过多更改。通常，游戏引擎被设计为基于组件的架构系统，允许使用更专用（并且通常更昂贵）的其他中间件来替换或扩展引擎中的特定系统，如用于物理模拟的 Havok，用于声音的 Miles Sound System 或 Bink 视频，用于全局光照组件 Enlighten，用于植被模拟的 Speedtree。一些游戏引擎，如 RenderWare 甚至被设计为一系列松散连接的游戏中间件组件，可以选择性地组合以创建定制化引擎。

尽管名称中有游戏两个字，似乎看起来功能比较单一，但实际上游戏引擎通常也能用于有实时图形需求的其他类型的交互式应用，如营销演示、建筑可视化、训练模拟等。

一些游戏引擎仅提供实时3D渲染能力，而没有游戏所需的全部其他的功能。这些引擎需要游戏开发者来实现其他功能或使用其他游戏中间件来组合成完整的引擎。这些类型的引擎通常被称为“图形引擎”、“渲染引擎”或“3D引擎”，而不是包含更多组件的术语所谓“游戏引擎”。图形引擎的一些示例是：Crystal Space、Genesis3D、Irrlicht、OGRE、RealmForge、Truevision3D和Vision Engine。

现代游戏引擎是一整套复杂的应用程序，通常有几十个子系统交互，以确保精确控制用户体验。游戏引擎在持续演变，在渲染、脚本、美术和关卡设计之间形成了更大的独立性。例如，在如今的一个典型的游戏开发团队中，美术人员的数量是程序人员数量的许多倍。

随着移动平台的爆发，更多的游戏引擎出现，其中最重要的便是Unity，它也是第一个推出跨平台的商用引擎：写一遍代码，引擎通过自己的翻译器，生成CIL（通用中间语言），运行时从CIL到本地设备即时编译运行。这种方式大大省去了游戏开发者在移动平台两大系统（安卓和iOS）间的支持工作。随后，Unreal Engine（虚幻引擎）也慢慢开始跨平台，除了老本行PC和游戏机，最近几年也开始支持移动平台。

游戏引擎也随着时间进化更新着，对于新技术的出现也会很快吸收并转化为引擎自身的部分。随着新的图形API的推出，主流商业引擎也纷纷开始投入人力进行DirectX 12和Vulkan的支持，并且对于新兴的VR产业，也不断进行优化来提供更好的性能以帮助VR开发者，Unreal Engine最近就优化了它的前向渲染，使得开发者可以获得更多的性能。

随着游戏引擎技术的成熟和变得更加用户友好，游戏引擎的应用范围已经扩大。以CryEngine为例，它们现在被用于更加严肃的领域：可视化、培训、医疗和军事模拟应用程序。同时为了促进这种引擎使用范围的扩张，包括移动电话（如Android手机、iPhone）和网络浏览器等硬件、软件在内的新平台，都变成了引擎的目标平台（如WebGL、Shockwave、Flash、Trinigy的WebVision、Silverlight、Unity Web Player、O3D和纯DHTML）。

此外，更多的游戏引擎正建立在诸如Java和C#/.NET（如TorqueX和Visual3D.NET）、Python（Panda3D）或Lua Script（Leadwerks）等高级语言之上。由于大多数3D游戏现在主要是GPU限制的（即受到显卡能力的限制），所以更高级语言的翻译开销导致的潜在性能降低变得可以忽略，而由这些语言提供的生产力增益则是巨大的。最近的这些趋势被微软等公司用来推动独立游戏开发。微软开发的XNA便是作为在Xbox和相关产品上发布的所有游戏的首选SDK，这里包括Xbox Live Indie游戏频道，其是专为那些没有足够资源在零售货架上包装销售游戏的小型开发商而设计的。

游戏引擎、D3D/OpenGL、Shader、编程、显卡是游戏开发环节上的整体生态，互相关联、互相影响。显卡的革新和技术往往会影响其他4个形态，然后最终影响游戏和图形应用开发者的内容质量。一个好的开发者应该对这几个模块都有所理解。

AMD Radeon RX480显卡是一款非常适合读者学习开发的中端显卡。后面的章节展示了一些在Unity引擎上的Shader编程，所有程序都由AMD员工在该显卡上一一验证，以方便读者学习。

1.4 游戏引擎中的 Shader 编程

当前的3D内容制作大多数都不会通过直接的3D API（OpenGL、DirectX等）实现，而是通过游戏引擎这样的中间件实现（包括各种商业引擎和公司自研引擎）。对于游戏引擎来说，Shader的编写与GLSL也有较大的不同。目前主流的游戏引擎对于Shader的编写主要需要考虑以下两个因素。

（1）跨平台。

游戏引擎生成的Shader程序必须保证在引擎支持的各大硬件平台上运行，不仅包括GLSL、HLSL等，还要保证各个移动平台上的行为一致性。各大引擎都使用了自己的解决方案，包括自定义的中间语言，HLSL到GLSL的解释器等。

而Vulkan API的推出可能会让这件事情变得相对简单，由于Vulkan定义了Shader通用的中间语言SPIR－V，因此引擎只要保证将自定义的语言生成正确的SPIR－V中间语言，就可以保证在各大支持Vulkan的硬件平台上的一致性。

（2）易用性。

在当前支持各种次世代渲染特性的引擎中，一个简单的、能用的Dhader往往需要支持引擎里面的各种渲染管线和各种配置，比如前向/后向渲染、有无阴影、高质量/低质量、各种不同的Shading Model BRDF（双向反射分部函数）等。大部分主流引擎都通过各种各样的自定义宏或函数等将常用的功能封装好，如在Unity的ShaderLab语言中可以通过#multicompile定义Shader应用的管线，而在UE4中甚至可以用UI指定每个Shader的特性。

1. Unity引擎中的Shader编程范例

Unity引擎中的Shader编程在语法上类似于CG语言，Unity除了可以支持编写通常的Vertex/Fragment shader，Compute shader等，还支持一种相对简单的材质Shader：Surface Shader。

2. 编写一个燃烧材质的Surface Shader

Surface Shader是Unity中生成物体材质用的受光照shader的低层Vertex/Fragement program的一种手段，我们只要指定一个SurfaceOutput的数据结构中的输出，就可以得到一个基于Unity5的新的基于物理光照模型的材质，并且让它在Unity所有的渲染管线中都起作用。接下来我们编写一个Surface Shader来实现简单的Normal Mapping。

首先在Unity中新建一个Standard Surface Shader，在Visual Studio中打开之后我们可以看到：

```
Shader"Custom/SurfaceTest" {
    Properties {
        _Color("Color",Color) = (1,1,1,1)
        _MainTex("Albedo(RGB)",2D) = "white" {}
        _Glossiness("Smoothness",Range(0,1)) = 0.5
        _Metallic("Metallic",Range(0,1)) = 0.0
    }
    SubShader {
```

```
        Tags { "RenderType" = "Opaque" }
        LOD 200

        CGPROGRAM
        #pragma surface surf Standard fullforwardshadows
        #pragma target 3.0

        sampler2D _MainTex;

        struct Input {
            float2 uv_MainTex;
        };

        void surf(Input IN, inout SurfaceOutputStandard o) {
        ......
        }
        ENDCG
    }
    FallBack "Diffuse"
```

我们可以看到surf函数就是我们要修改的输出，为了达到物体溶解的效果，我们会采用一张噪声纹理作为燃烧的纹理，同时用clipAPI去掉已燃烧掉部分的像素。

（1）给Shader添加一个纹理作为输入。

```
Properties{
_BurnTex("Burn Texture(RGB)",2D) = "white" {}
}
```

然后定义这个纹理的sampler：

```
sampler2D _BurnTex;
```

（2）给Shader添加一个float参数作为燃烧值。

```
_BurnAmount("Burn Amount",Range(0.0,1.0)) = 0.5
```

定义变量：

```
float _BurnAmount;
```

（3）根据纹理Clip掉燃烧的像素。

```
void surf(Input IN, inout SurfaceOutput o) {
        fixed3 burn = tex2D(_BurnTex, IN.uv_BurnTex);
        clip(burn.r - _BurnAmount);
    ......
}
```

这里我们根据 uv 值取出这一点燃烧纹理的值（这张纹理是灰度图，因此 RGB 值一样）然后用这个值减去_BurnAmount 参数，这样我们就可以通过调整 BurnAmount 参数来实现按照纹理逐步燃烧消失的效果了。

（4）给燃烧效果添加边缘火焰颜色。

我们定义两个颜色，作为火焰的初始颜色和结束颜色，并定义燃烧边缘的宽度作为 Shader 参数，根据燃烧的宽度对两个颜色进行插值作为当前像素的火焰颜色。

```
[HDR]_BurnFirstColor("Burn First Color",Color) = (1,0,0,1)
[HDR]_BurnSecondColor("Burn Second Color",Color) = (1,0,0,1)
_LineWidth("Burn Size",Range(0.0,0.2)) = 0.1
fixed4 _BurnFirstColor;
fixed4 _BurnSecondColor;
float _LineWidth;
void surf(Input IN,inoutSurfaceOutput o){
fixed t = 1 - smoothstep(0.0,_LineWidth,burn.r - _BurnAmount);
fixed3burnColor = lerp(_BurnFirstColor,_BurnSecondColor,t);
burnColor = pow(burnColor,10);
.....
```

注意，这里我们给燃烧颜色加上了 HDR 标记，是为了让颜色在支持 HDR 的渲染流程中能够具有发光的效果。

我们可以再定义一个 Shader 的主要纹理作为材质本身的贴图，并将燃烧颜色叠加上原来的颜色作为 Shader 最后的输出颜色。

```
_MainTex("Texture(RGB)",2D) = "white" {}
sampler2D _MainTex;
fixed3finalColor = lerp(albedo,burnColor,t * step(0.0001,_BurnAmount));
```

最后得到的效果如图 1-4 所示。

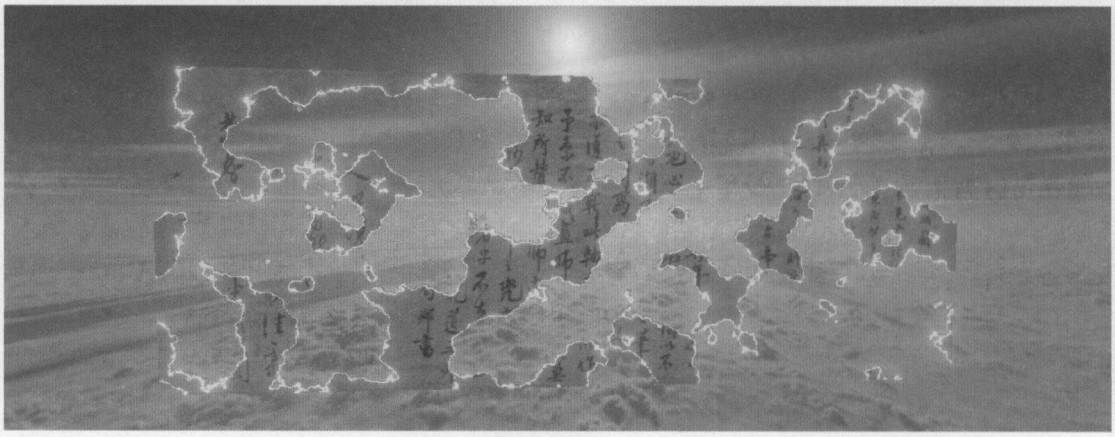


图 1-4　效果图

3. 使用 Vertex/Fragment Shader 编写一个描边材质

我们再来看一个使用 Vertex/Fragment Shader 加上一些 Unity 自定义的工具函数实现一个游戏中很常用的选中描边材质的效果。

这个材质的思路实现在于画两个 Pass，在第 1 个 Pass 时，我们剔除物体的背面，而在第 2 个 Pass 时，我们先在 Vertex Shader 中把顶点在投影空间沿着边缘的方向微微扩张，然后再次剔除物体正面，这样留下来的部分就是我们所需要的物体描边。

在 Unity 的 Vertex/Fragment 可编程 Shader 中，我们可以用类似于：

```
#pragma vertex vert
#pragma fragment frag
```

这样的语句来定义每个 Pass 的 vertex 和 fragment 函数。

先看第 1 个 Pass：

```
Pass
{
    Name "BASE"
    Cull Back
    Blend OneMinusSrcColor One
}
```

这个 Pass 走默认的渲染管线并且只做了背面剔除的工作。

重点在第 2 个 Pass：

```
    Pass
    {
        Name "OUTLINE"
        Tags{"LightMode" = "Always" }
        Cull Front
    Blend One Zero
    ......
CGPROGRAM
#pragma vertex vert
#pragma fragment frag
#include"UnityCG. cginc"
structappdata
{
    float4 vertex : POSITION;
    float3 normal : NORMAL;
};
......
uniformfloat _Outline;
uniformfloat4 _OutlineColor;
v2f vert(appdata v){
        v2f o;
```

```
            o. pos = mul( UNITY_MATRIX_MVP, v. vertex) ;
            float3 norm = mul( (float3x3) UNITY_MATRIX_IT_MV, v. normal) ;
            float2 offset = TransformViewToProjection( norm. xy) ;
            o. pos. xy += offset * _Outline;
            return o;
    }
    ......
    }
```

这里我们在 vertex 程序中需要得到的是最后投影空间的坐标所需要偏移的方向，对于每一个要着色的顶点来说，我们先将法线乘以 Model - view 矩阵的逆矩阵的转置，这会将法线由物体空间变换到我们的视空间（Eye Space），紧接着调用 Unity 内置的由视空间到投影空间的函数我们就可以得到在投影空间所需要扩张的方向，最后把这个二维向量加到变换到投影空间的顶点位置的 *xy* 分量即可（也就是沿着屏幕的方向扩张）。

在 fragment 程序中，我们便可以任意指定想要的描边颜色。

最后得到的效果如图 1-5 所示。

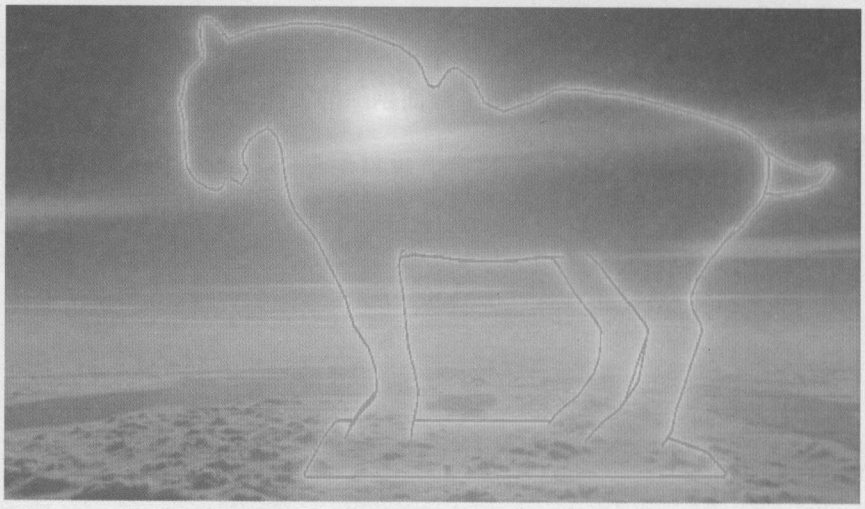


图 1-5　效果图

1.5　Vulkan 介绍

如果你现在使用或者开始学习 OpenGL 编程，那你一定要了解 OpenGL 的下一代标准——Vulkan，这是未来 Khronos 不遗余力推动的下一个版本。它由 AMD Mantle 接口演变而来，同时微软 DirectX 12 也由 Mantle 演变，而 DirectX 12 则是微软在 Windows 10 力推的标准。Vulkan 是一个底层 3D 图形 API，允许开发者获得硬件底层控制能力，同时减少性能开销，Vulkan 为开发人员提供通常留给驱动程序的控制能力，如线程管理、内存管理和错误检查等功能。目前 Vulkan 1.0 标准已经完成并正式发布。上一代的 OpenGL/ES 并不会被遗弃，还会继续发展，很有可能 OpenGL/ES 变为 Vulkan 的简化 API。

与OpenGL/ES相比，Vulkan的优势如下。

（1）更简单的显示驱动层。

Vulkan提供了能直接控制和访问底层GPU的显示驱动抽象层，显示驱动只是对硬件薄薄的封装，这样能够显著提升操作GPU硬件的效率和性能。之前OpenGL的驱动层对开发者隐藏的很多细节，现在都暴露出来了。Vulkan甚至不包含运行期的错误检查层。驱动层干的事情少了，隐藏的Bug也就少了。

（2）支持多线程。

OpenGL的效率一直以来被诟病，而且一直没有改变。Vulkan不再使用OpenGL的状态机设计，内部也不保存全局状态变量。显示资源完全由应用层负责管理，包括内存管理、线程管理、多线程绘制命令产生、渲染队列提交等。应用程序可以充分利用CPU的多核多线程的计算资源，减少CPU等待，降低延迟。带来的问题是，线程间的同步问题也由应用程序负责，从而对开发人员的要求也更高。

Vulkan在设计之处就考虑多线程问题，可以说它就是为了多线程而设计的。

命令缓冲和和命令调度队列是Vulkan支撑多线程的重要组成部分，类似OpenGL的上下文状态。

Vulkan的命令缓冲都是独立的、互不干扰的单元，支持在多个线程中创建。这样可以由多个线程创建不同的绘制命令，由单独的线程管理渲染命令队列，统一提交给GPU绘制。

（3）预编译Shaders。

驱动层不提供前端Shader编译器，只支持标准可移植中间表示二进制代码（SPIR－V）。这样既提高了执行Shaders的效率，又增加了将来着色语言的灵活性。所以目前的GLSL/HLSL可以直接通过工具转换为SPIR－V，在Vulkan中使用。这样就可以使用离线的Shader编译。

另外，SPIR－V还支持OpenCL！

（4）跨平台。

支持桌面、移动设备、游戏主机、嵌入式……只要需要显示的地方，貌似都能支持。

这也是Vulkan与DirectX12相比的优势。

（5）Vulkan窗口系统集成。

Vulkan把显示设备的创建和窗口系统的创建明确分开。显示设备只提供支持可视化的显示队列的接口，应用程序控制如何显示绘制结果。比如前后显示缓冲这些的都由应用程序创建和设置。

Vulkan提供标准的扩展API支持多窗口系统，如Android、Windows、X等窗口系统。

未来几年Vulkan和DirectX 12会被微软、Google、Valve、Intel、AMD、Nvidia等诸多巨头鼎力支持。GPU也会因为这些标准的树立和VR/AR的应用而越来越广泛被使用。

第2章 GLSL 语言

2.1 变量

GLSL 语言是 OpenGL Shading Language 的简称。GLSL 的变量命名方式与 C 语言类似。变量的名称可以是字母、数字及下画线，但变量名不能以数字开头，GLSL 的内部变量都是以 gl_ 作为前缀的，因此自定义的变量名不能以 gl_作为前缀。还有一些 GLSL 保留的名称不能够作为变量的名称。GLSL 的变量类型除了常用的布尔型、整型、浮点型基本类型外，GLSL 也支持在并行着色器上能够做并行计算的一些变量类型。GLSL 语言和 C 语言等非并行语言的主要区别之一就是变量类型支持向量、矩阵等可以进行并行操作的类型。GLSL 的基本类型见表 2-1。

表 2-1 GLSL 的基本类型

类 型	描 述
void	跟 C 语言的 void 类似，表示空类型；作为函数的返回类型，表示这个函数不返回值
bool	布尔类型，可以是 true 和 false，以及可以产生布尔型的表达式
int	整型，代表至少包含 16 位的有符号的整数；可以是十进制的、十六进制的、八进制的
float	浮点型
bvec2	包含 2 个布尔成分的向量
bvec3	包含 3 个布尔成分的向量
bvec4	包含 4 个布尔成分的向量
ivec2	包含 2 个整型成分的向量
ivec3	包含 3 个整型成分的向量
ivec4	包含 4 个整型成分的向量
mat2 mat2x2	2 ×2 的浮点数矩阵类型
mat3 mat3x3	3 ×3 的浮点数矩阵类型
mat4x4	4 ×4 的浮点矩阵
mat2x3	2 列 3 行的浮点矩阵，OpenGL 的矩阵是列主顺序的
mat2x4	2 列 4 行的浮点矩阵
mat3x2	3 列 2 行的浮点矩阵
mat3x4	3 列 4 行的浮点矩阵
mat4x2	4 列 2 行的浮点矩阵
mat4x3	4 列 3 行的浮点矩阵
sampler1D	用于内建的纹理函数中引用指定的 1D 纹理的句柄，只可以作为一致变量或者函数参数使用

续表

类　型	描　述
sampler2D	二维纹理句柄
sampler3D	三维纹理句柄
samplerCube	cube map 纹理句柄
sampler1DShadow	一维深度纹理句柄
sampler2DShadow	二维深度纹理句柄

2.2　结构体

为了方便使用，需要把几个变量组合起来，这样的话就要结构体。一个结构体包含几个用户自定义的基本类型。结构体至少包含一个成员。固定大小的数组也可以被包含在结构体中。GLSL 的结构体不支持嵌套定义。只有预先声明的结构体可以嵌套其中。如以下程序所示，自定义一个光的结构体类型，包含位置和颜色两种基本类型。

```
struct Light
{
    vec3 position;
    vec3 color;
};
```

可以通过 = 为结构体赋值，或者使用 == 、! = 来判断两个结构体是否相等。只有结构体中的每个成分都相等，那么这两个结构体才是相等的。

```
Light light0;
Light light1;
light0 == light1;
```

访问结构体的内部成员使用“点”号来访问，程序如下。

```
light0. position
```

数组

GLSL 中只可以使用一维的数组。数组的类型可以是一切基本类型或结构体。指定显示大小的数组可以作为函数的参数或者是返回值，也可以作为结构体的成员。数组类型内建了一个 length( ) 函数，可以返回数组的长度。

2.3　修饰符

修饰符是 GLSL 语言和 C 语言的一个重要的区别，有很多特定的用于 GPU 上的修饰符。也就是虽然变量的类型一样，但是修饰符不一样，那么变量的含义和使用的方式也不一样。GLSL 语言的修饰符用于从外部向 GLSL 传递数据，或者 GLSL 着色器之间传递数据。通过修饰符才能够更好地明确变量应该如何被使用。变量声明的修饰符见表 2-2。

表 2-2　变量声明的修饰符

修 饰 符	描　　述
const	常量修饰符是只读不可修改的，常量值必须在声明时进行初始化
attribute	表示只读的顶点数据，只用在顶点着色器中。数据来自当前的顶点状态或顶点数组。必须是全局范围声明的，不能在函数内部。一个 attribute 可以是浮点数类型的标量、向量或者矩阵。不能是数组或结构体
uniform	一致变量。在着色器执行期间一致变量的值是不变的。与 const 常量不同的是，这个值在编译时期是未知的是由着色器外部初始化的。一致变量在顶点着色器和像素着色器之间是共享的。只能在全局范围进行声明
varying	顶点着色器的输出。例如，颜色或纹理坐标，插值后的数据作为像素着色器的只读输入数据。必须是全局范围声明的全局变量。可以是浮点数类型的标量、向量、矩阵。不能是数组或结构体。用于在顶点着色器和像素着色器之间传递数据
invariant	不变量修饰符。用于表示顶点着色器的输出和任何匹配像素着色器的输入，在不同的着色器中计算产生的值必须是一致的。所有的数据流和控制流，写入一个 invariant 变量的是一致的。除非必要，不要使用这个修饰符
in	用在函数的参数中，表示这个参数是输入的，在函数中改变这个值，并不会影响对调用的函数产生副作用。这个是函数参数默认的修饰符
out	用在函数的参数中，表示该参数是输出参数，值是会改变的
inout	用在函数的参数，表示这个参数即是输入参数也是输出参数

这里面最重要的是 attribute、uniform、varying 修饰符。uniform 修饰符用来从外部给 GLSL 程序传递参数，attribute 修饰符用来设置每个顶点的数据，varying 修饰符用来在顶点着色器和像素着色器之间传递数据。

2.4　内置变量

虽然可以用 attribute 属性来给每个顶点传递数据，但 GLSL 也有很多内置变量。这些变量方便了数据的传递。内置变量可以与固定函数功能进行交互。在使用前不需要声明。这些内置变量大多数和 OpenGL 相关联。通过 OpenGL 外部来进行设置，然后自动赋值给 GLSL 的内置变量。但这些内置变量如果没有在 OpenGL 中进行赋值，那么内置变量的值为空。例如，纹理坐标，如果 OpenGL 中没有设置纹理坐标，那么纹理坐标内置变量就无法得到数值。内置变量都是以 gl 符号开头的变量。因此自定义的变量要避免采用 gl 作为开头符号。

顶点着色器可用的内置变量见表 2-3。

表 2-3　顶点着色器可用的内置变量

名　　称	类　　型	描　　述
gl_Color	vec4	输入属性，表示顶点的主颜色
gl_SecondaryColor	vec4	输入属性，表示顶点的辅助颜色
gl_Normal	vec3	输入属性，表示顶点的法线值
gl_Vertex	vec4	输入属性，表示物体空间的顶点位置
gl_MultiTexCoordn	vec4	输入属性，表示顶点的第 n 个纹理的坐标
gl_FogCoord	float	输入属性，表示顶点的雾坐标

续表

名　称	类　型	描　述
gl_Position	vec4	输出属性，变换后的顶点的位置，用于后面的固定的裁剪等操作。所有的顶点着色器都必须写这个值
gl_ClipVertex	vec4	输出坐标，用于用户裁剪平面的裁剪
gl_PointSize	float	点的大小
gl_FrontColor	vec4	正面主颜色的 varying 输出
gl_BackColor	vec4	背面主颜色的 varying 输出
gl_FrontSecondaryColor	vec4	正面辅助颜色的 varying 输出
gl_BackSecondaryColor	vec4	背面辅助颜色的 varying 输出
gl_TexCoord[ ]	vec4	纹理坐标的数组 varying 输出
gl_FogFragCoord	float	雾坐标的 varying 输出
gl_ModelViewMatrix	Mat4	模型视图变换矩阵
gl_ProjectMatrix	Mat4	投影矩阵
gl_ModelViewProjectMatrix	Mat4	模型视图投影变换矩阵
gl_NormalMatrix	Mat4	法向量变换到视空间矩阵
gl_TextureMatrix[gl_MatTextureCoords]	Mat4	各纹理变换矩阵

像素着色器常用内置变量见表2-4。

表2-4　像素着色器常用内置变量

名　称	类　型	描　述
gl_Color	vec4	包含主颜色的插值只读输入
gl_SecondaryColor	vec4	包含辅助颜色的插值只读输入
gl_TexCoord[ ]	vec4	包含纹理坐标数组的插值只读输入
gl_FogFragCoord	float	包含雾坐标的插值只读输入
gl_FragCoord	vec4	只读输入，窗口的 x，y，z 和 1/w
gl_FrontFacing	bool	只读输入，如果是窗口正面图元的一部分，则这个值为 true
gl_PointCoord	vec2	点精灵的二维空间坐标范围在（0.0，0.0）到（1.0，1.0）之间，仅用于点图元和点精灵开启的情况下
gl_FragData[ ]	vec4	使用 glDrawBuffers 输出的数据数组，不能与 gl_FragColor 结合使用
gl_FragColor	vec4	输出的颜色用于随后的像素操作
gl_FragDepth	float	输出的深度用于随后的像素操作，如果这个值没有被写，则使用固定功能管线的深度值代替

2.5　操作符和构造函数

1. 操作符

GLSL语言是一种编程语言，因此需要操作符和构造函数。GLSL语言的操作符与C语言相似。GLSL不支持求地址的和解引用的操作符，因为GLSL不能直接操作地址；也不支持类型转换操作。操作符优先级从高到低排列见表2-5。

表 2-5　操作符优先级

操　作　符	描　　述
( )	用于表达式组合，函数调用，构造
[ ]	数组下标，向量或矩阵的选择器
.	结构体和向量的成员选择
++ --	前缀或后缀的自增、自减操作符
+ - !	一元操作符，表示正、负、逻辑非
* /	乘、除操作符
+ -	二元操作符，表示加、减操作
< > <= >= == !=	小于、大于、小于或等于、大于或等于、等于、不等于判断符
&& || ^^	逻辑与、或、异或
?:	条件判断符
= += -= * = / =	赋值操作符
,	表示序列

2. 构造函数

GLSL 的构造函数和 C 语言类似，特别的地方是矩阵。对于矩阵，OpenGL 中矩阵是列主顺序的。如果只传了一个值，则会构造成对角矩阵，其余的元素为 0。代码如下。

```
mat3 m3 = mat3(1.0)
```

构造出来的矩阵如下。

```
1.0 0.0 0.0
0.0 1.0 0.0
0.0 0.0 1.0
```

下面都是合法的矩阵构造函数。

```
mat2 matrix3 = mat2(1.0);
mat2 matrix4 = mat2(mat4(2.0));
mat2 matrix1 = mat2(2.0,0.0,0.0,2.0);
mat2 matrix2 = mat2(vec2(2.0,0.0),vec2(0.0,1.0));
```

3. 元素选择

一个向量中的分量可以通过下面几种方法选择，这几种方法是等价的。

```
{x,y,z,w}
{r,g,b,a}
{s,t,p,q}
```

这 3 种方法通常分别用于顶点、颜色、纹理坐标。之所以提供 3 种方法，是为了方便识别。这 3 种方法是等价的，但每种方法里的 4 个成分不可用互相混用。下面是用法举例。

```
vec3 test = {0.1,0.2,0.3};
float test0 = test.r;
vec2 test1 = test.yz;
```

向量中的元素可以重复和颠倒顺序，代码如下。

```
vec3 test4 = test. yxz;
vec4 test6 = test. sst;
```

赋值时可以按照选择顺序进行，代码如下。

```
vec4 test = {0.0,1.0,2.0,3.0}
test. x = -1.0;
test. yx = vec2(1.0,2.0);
test. zx = vec2(1.0,2.0);
```

下面的用法是不合法的。

```
float test1 = test. q
vec2 test1 = test. ry
```

上面第1个操作会造成越界，因为test变量只有3个值。第2个操作把两种方法的成分混用了。

向量或矩阵中的元素也可以通过下标来访问，代码如下。

```
float test1 = test[1]
```

在矩阵中，可以通过一维的下标来获得该列的向量，还可以通过二维的下标来获得向量中的元素。

```
mat3 test = mat3(1.0)
vec3 test1 = test[0]
float test2 = test[0][0]
```

4. 控制流

和C语言一样，GLSL也有for、while、do/while、if/else的循环方式。可以用continue跳入下一次循环，用break结束循环。Discard是像素着色器中的一种特殊的控制流。使用discard会退出像素着色器，不执行后面的像素着色操作。像素也不会写入帧缓冲区。

5. 函数

在每个Shader中必须有一个main函数。main函数中的void参数是可选的。GLSL中的函数，必须是在全局范围定义和声明的，不能在函数定义中声明或定义函数。函数必须有返回类型，参数是可选的。参数的修饰符是可选的。结构体和数组也可以作为函数的参数。如果是数组作为函数的参数，则必须制定其大小。在调用传参时，只传数组名就可以了。GLSL的函数支持重载。函数可以同名但其参数类型或参数个数不同。

```
void main(vod)
{
……
}
```

2.6 内置函数

GLSL 内置了一些函数，这些函数可以完成一些常用的操作。这样就不需要为每个基本的功能都重写函数了。这些内置函数大部分都和 C 语言里的函数类似，但有一些是特殊用于 GPU 并行计算的。其中常用的内置函数见表 2-6。

表 2-6 常用的内置函数

名　称	描　述
pow( )	幂函数（对矢量和标量同样有效，下同）
exp( ),log( )	指数函数，对数函数
abs( )	绝对值
sqrt( )	平方根
max( ),min( )	最值
ceil( ),floor( )	取大于实参的最小整数，取小于实参的最大整数
sin( ),cos( ),tan( )	三角函数
asin( ),acos( ),atan( )	反三角函数
sinh( ),cosh( ),<a>tanh( )	双曲正弦，双曲余弦，双曲正切（以及相应的反函数）
length( )	向量长度
distance( )	两个向量的距离
dot( ),cross( )	数乘，叉乘
matrixCompMult( )	矩阵对应元素分别相乘
transpose( ),determinant( ),inverse( )	矩阵的转置，行列式，逆
lessThan( ),greaterThan( ),equal( )	小于，大于，等于（对实参向量对应位置的每个分量做大小比较，生成布尔向量）
Degrees( ),radians( )	角度函数，弧度函数
mod( )	取模
clamp(x,y,z)	确保一个值在特定范围内，返回 min(max(x,y),z)
mix(x,y,a);	使用 a 对 x，y 执行线性混合，返回 x(1-a)+y*a
step( )	第二个参数小于或等于第一个参数则返回 0，否则返回 1
smoothstep(a,b,x)	如果 x<=a，则返回 0；如果 x>=b，则返回 1；当 a<x<b 时，就在 0 与 1 之间进行平滑的 Hermite 插值。
normalize( )	返回长度为 1 的矢量
reflect( )	根据入射矢量 I 和查看放向的矢量 N 计算反射矢量 R，$R=I-2*\text{dot}(N,I)*N$
texture1D( ),texture2D( ),texture3D( ),	根据纹理对象和纹理坐标，返回一维、二维、三维纹理值
dFdx(p)	返回输入参数 p 在 x 上的导数
dFdy(p)	返回输入参数 p 在 y 上的导数
fwidth(p)	返回输入参数 p 在 x 和 y 上导数绝对值的和

第 3 章 GLSL 框架设计

3.1 加载和编译

GLSL 程序本身是个文本文件，需要 OpenGL 的函数来加载和编译后才能够使用。文本文件的后缀名一般设置为 Vert 和 Frag，只是为了方便识别，并不是强制要求，也可以根据习惯写成其他的后缀名。在 C#中，需要相关 OpenGL 函数来调用 GLSL 的程序。OpenGL 提供了一些相应的函数，来加载编译外部的 GLSL 程序。首先分别加载和创建顶点着色器和片段着色器的程序，然后在内存里把两个着色器连接起来构成一个完整的 GLSL 渲染程序。整个 GLSL 程序加载编译的流程图如图 3-1 所示。

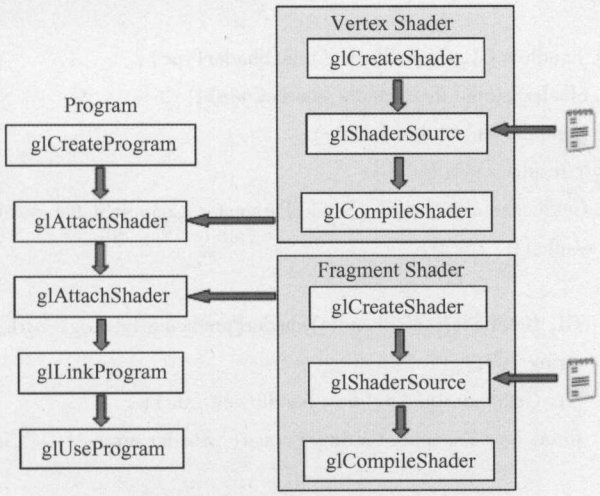


图 3-1 GLSL 程序加载编译流程图

第一步：加载编译 GLSL 程序。

GLSL 的渲染器（Shader）分为 3 种，枚举类型如下。本书中只用到两种。对第 3 种不做讨论。

```
public enum ShaderType
    {
        FragmentShader = 35632,
        VertexShader = 35633,
```

```
        GeometryShaderExt = 36313,
    }
```

（1）GL. CreateShader

这个函数用来生成一个顶点，或者像素类型的空的渲染器。

（2）GL. ShaderSource

GLSL 程序的具体内容是存在于硬盘的文本文件，需要先加载到内存的字符串，然后使用这个函数设置渲染器的内容。

（3）GL. CompileShader

编译 GLSL 程序。

完整的代码如下。

```
public AbstractShader(string file)
        {
            this.file = file;
            string sourceCode;
            using (StreamReader reader = File.OpenText(this.file))
            {
                sourceCode = reader.ReadToEnd();
            }
            this.handle = GL.CreateShader(this.ShaderType);
            GL.ShaderSource(this.handle,sourceCode);
            GL.CompileShader(this.handle);
            int[] results = new int[1];
            GL.GetShader(this.handle,ShaderParameter.CompileStatus,results);
            if (results[0] == 0)
            {
                GL.GetShader(this.handle,ShaderParameter.InfoLogLength,results);
                string info;
                GL.GetShaderInfoLog(this.handle,out info);
                throw new Exception(string.Format("Shader error:{0}",info));
            }
        }
```

上述函数的输入字符串是 GLSL 程序的文件名，执行完毕后，GLSL 程序就被 OpenGL 加载和编译了。顶点和像素着色器都可以用上述代码进行加载和编译，从中可以看出，GLSL 代码不需要必须是硬盘上的文本文件，也可以直接在内存生成。

第二步：连接 GLSL 程序。

在顶点和像素着色器程序加载和编译后，在程序里是独立存在的，需要用 OpenGL 的函数把它们连接到一个程序里面。用到的 OpenGL 函数如下。

（1）GL. CreateProgram

（2）GL. AttachShader

(3) GL. LinkProgram

完整的程序如下所示。

```
private void LinkProgram( )
    {
        this. handle = GL. CreateProgram( ) ;
        GL. AttachShader( this. handle , this. vertexShader. handle) ;
        if ( FSFileName ! = string. Empty)
        {
            GL. AttachShader( this. handle , this. fragmentShader. handle) ;
        }

        GL. LinkProgram( this. handle) ;
        vertexShader. Dispose( ) ;
        if ( FSFileName ! = string. Empty)
        {
            fragmentShader. Dispose( ) ;
        }
        CheckAfterLink( ) ;
    }
```

第三步：使用。

在完成上述步骤之后，需要开启使用这个 GLSL 程序，代码如下。

```
public void Use( )
    {
        GL. UseProgram( this. handle) ;
    }
```

如果需要停止使用 GPU，也就是采用 OpenGL 内置的渲染功能，可以调用下面的函数。

```
public void DisableGPUProgram( )
{
    GL. UseProgram(0) ;
}
```

第四步：参数传递。

GLSL 程序需要通过 OpenGL 来把参数从外部传递进来，要用到的 OpenGL 函数如下。

(1) GL. GetUniformLocation

这个函数得到内存中 GLSL 用到的外部参数变量的位置。

(2) GL. Uniform1

(3) GL. Uniform2

(4) GL. Uniform3

(5) GL. Uniform4

具体把变量的值传递进 GLSL 程序，根据变量类型的不同，可以分为以上 4 种。

使用方法代码如下。

```
GL. Uniform1( GL. GetUniformLocation( this. handle, paramter) , value)
```

3.2 程序架构

1. 程序继承

因为所有 GLSL 程序使用时加载编译链接的方法都一样，不一样的只是具体程序的文件名和参数，因此可以设置一个基类，其他的程序都可以从此基类继承，从而可以把共同的功能函数放到基类里面。AbstractShader 类是所有各种 GLSL 渲染程序的抽象基类，其他程序都是由此程序继承得到的。例如，下面的 Gooch 渲染程序继承了 AbstractShader 类。

```
public class GPURenderGooch : AbstractGPURender
{
    public GPURenderGooch( )
    {
        VSFileName = "GLSL/NPR/Gooch. vert";
        FSFileName = "GLSL/NPR/Gooch. frag";
    }
    public override void PassParameter( )
    {
        GL. Uniform3( GL. GetUniformLocation( this. handle, "LightPosition") ,
                      ConfigGPUCommon. Instance. GetLightPosition( ) ) ;
        GL. Uniform3( GL. GetUniformLocation( this. handle, "SurfaceColor") ,
                      ConfigGPUGooch. Instance. GetSurfaceColor( ) ) ;
        GL. Uniform3( GL. GetUniformLocation( this. handle, "WarmColor") ,
                      ConfigGPUGooch. Instance. GetWarmColor( ) ) ;
        GL. Uniform3( GL. GetUniformLocation( this. handle, "CoolColor") ,
                      ConfigGPUGooch. Instance. GetCoolColor( ) ) ;
        GL. Uniform1( GL. GetUniformLocation( this. handle, "DiffuseWarm") ,
                      ConfigGPUGooch. Instance. DiffuseWarm) ;
        GL. Uniform1( GL. GetUniformLocation( this. handle, "DiffuseCool") ,
                      ConfigGPUGooch. Instance. DiffuseCool) ;
    }
}
```

在此程序中，只要设定好相应的 GLSL 程序的文件名和传递的参数，其他的部分都是在父程序中进行的。

2. 参数设置类

GPU 程序中的参数需要通过外部的界面进行修改。可以采用把所有的参数都放在一个参数设置类 ConfigGPUBase 里面，这个类可以是单实例的，这样的话，就可以方便参数的传递。和上述 GLSL 程序相关的参数设置类如下。

```
public class ConfigGPUGooch : ConfigGPUBase
{
    private static ConfigGPUGooch singleton = new ConfigGPUGooch();
    public static ConfigGPUGooch Instance
    {
        get
        {
            if (singleton == null)
                singleton = new ConfigGPUGooch();
            return singleton;
        }
    }
    private Color surfaceColor = Color.FromArgb(255,191,191,191);
    private Color warmColor = Color.FromArgb(255,153,153,153);
    private Color coolColor = Color.FromArgb(255,0,0,153);
    private float diffuseWarm = 0.9f;
    private float diffuseCool = 0.1f;
```

为了方便通过界面设置，需要在参数设置类里面把参数包装为属性，这样就可以通过 C# 的属性控件来直接修改了，代码如下。

```
public Color SurfaceColor
    {
        get
        {
            return surfaceColor;
        }
        set
        {
            surfaceColor = value;
        }
    }

    public Color WarmColor
    {
        get
        {
            return warmColor;
        }
        set
        {
            warmColor = value;
```

```
public Color CoolColor
    {
        get
        {
            return coolColor;
        }
        set
        {
            coolColor = value;
        }
    }
    public float DiffuseWarm
    {
        get
        {
            return diffuseWarm;
        }
        set
        {
            diffuseWarm = value;
        }
```

```
        }
    }
```

```
    }
    public float DiffuseCool
    {
        get
        {
            return diffuseCool;
        }
        set
        {
            diffuseCool = value;
        }
    }
```

3. 参数设置界面

参数设置的界面类是 FormGPU，这个类主要采用了 PropertyGrid 控件，这个控件可以直接修改链接的参数。GLSL 程序参数设置界面如图 3-2 所示。

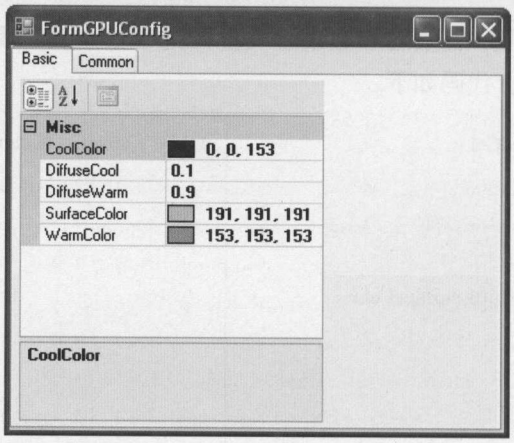


图 3-2　GLSL 程序参数设置界面

4. 程序切换

为了方便在各种 GPU 程序直接进行切换，设置一个总控制类 GlobalGPU。这个类负责删除前一个 GPU 程序，调用下一个 GPU 程序，同时也负责 GPU 参数的传递。外部程序并不直接调用具体的 GPU 程序，都是通过这个总控类来进行的。具体的代码如下。

```
public class GlobalGPU
    {
        private static GlobalGPU instance = null;
        public static GlobalGPU Instance
        {
            get
```

```
        {
            if (instance == null)
            {
                instance = new GlobalGPU();
            }
            return instance;
        }
    }
    public AbstractGPURender GPURender = null;
    public void SwitchGPURender(AbstractGPURender render)
    {
        if (GPURender != null)
        {
            GPURender.Dispose();
        }
        GPURender = render;
        if (GPURender != null)
        {
            GPURender.Init();
        }
        else
        {
            OpenGLManager.Instance.DisableGPUProgram();
        }
    }
    public void ResetGPURender()
    {
        if (GPURender != null)
        {
            GPURender.Use();
            GPURender.SetupParameter();
            GPURender.PassParameter();
        }
    }
    public void Dispose()
    {
        if (GPURender != null)
        {
            GPURender.Dispose();

            GPURender = null;
        }
```

```
                OpenGLManager. Instance. DisableGPUProgram( ) ;
            }
        }
```

5. 菜单显示

每个 GPU 渲染效果都对应一个菜单界面，可以通过单击菜单界面进行渲染效果切换。因为渲染效果很多，可以把这些渲染效果进行分类。然后每类里面的渲染效果放到相应的子菜单里面。菜单界面如图 3-3 所示。

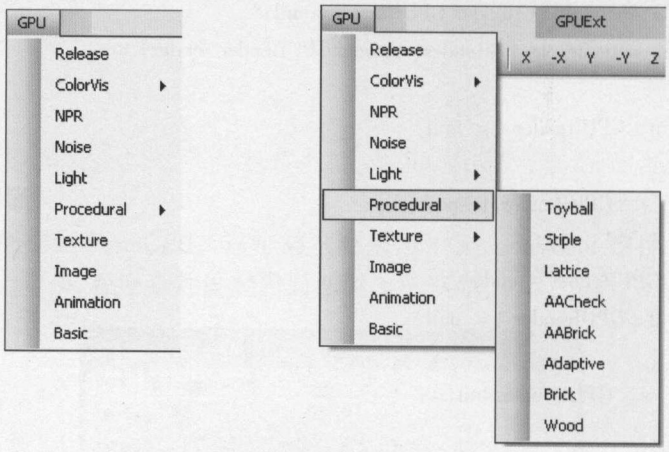


图 3-3　GPU 渲染菜单界面

菜单的加载是动态的，根据枚举类型里面的 GPU 渲染效果选项，动态生成菜单及每个菜单下面相应的子菜单。相应的代码如下。

（1）初始化菜单

```
public MenuGPURender( )
        {
            InitializeComponent( ) ;
            InitMenu( typeof( EnumGPUClass) , gPUToolStripMenuItem) ;
        }

        private void InitMenu( Type type, ToolStripMenuItem menu)
        {
            menu. Name = " toolToolStripMenuItem" ;
            foreach ( var typeItem in Enum. GetValues( type) )
            {
                ToolStripMenuItem item = new ToolStripMenuItem( ) ;
                item. Name = typeItem. ToString( ) + " ToolStripMenuItem" ;
                item. Text = typeItem. ToString( ) ;
                item. Tag = typeItem;
                item. MouseHover += GPU_Function;
```

```
            menu. DropDownItems. Add( item) ;
        }
    }
```

(2) 初始化子菜单

```
private void GPU_Function( object sender, EventArgs e)
    {
        ToolStripMenuItem menuItem = ( ToolStripMenuItem) sender;
        Type type = GPURenderCore. Instance. GetType( menuItem. Text) ;
        InitSubMenu( type, menuItem) ;
    }

    private void InitSubMenu( Type type, ToolStripMenuItem menu)
    {
        menu. DropDownItems. Clear( ) ;
        foreach ( var typeItem in Enum. GetValues( type) )
        {
            ToolStripMenuItem item = new ToolStripMenuItem( ) ;
            item. Name = typeItem. ToString( )  + " ToolStripMenuItem" ;
            item. Text = typeItem. ToString( ) ;
            item. Tag = typeItem;
            item. Click  += Click_Function;
            menu. DropDownItems. Add( item) ;
        }
    }
```

6. 类

在整个 GLSL 程序调用和使用的过程中，涉及如下几个类。

(1) GPURenderGooch
(2) AbstractShader
(3) FragementShader
(4) VertexShader
(5) AbstarctGPURedner
(6) ConfigGPUBase
(7) ConfigGPUGooch
(8) GlobalGPU
(9) FormGPU
(10) GPURenderCore
(11) MenuGPURender
(12) EnumGPUClass
(13) EnumAnimation

3.3 着色器简介

三维模型的渲染就是把三维模型及设置在这些模型上的属性，如颜色等参数变换为二维图像。OpenGL 本身提供了内置的渲染函数，也就是如何把三维模型变换为二维图像的算法已经内置到函数里了，只需要调用这些函数，并设置好参数就能够得到二维图像。这样的缺点是三维模型变换到二维图像的算法是固定的，无法改变，能改变的只是参数。例如，OpenGL 光照公式是固定死的，假如需要渲染出其他不同的光照效果，不适用 GLSL 的话，无法实现。GLSL 运行在 GPU 上，可以自定义光照公式，但使用 GLSL 进行渲染，OpenGL 本身的固定函数就无法使用，也就是必须实现原来 OpenGL 固定的光照函数，而不能只实现部分函数，同时还采用 OpenGL 的其他光照部分来混合调用。例如，OpenGL 本身内置了经过矩阵变换把三维顶点位置变换到二维，如果用 GLSL 来做，就需要实现这个矩阵变换过程。GLSL 程序是执行在 GPU 上的。三维数据输入到 GPU，然后经过着色器再变为二维像素。GLSL 程序主要分为两个模块：顶点着色器（Vertex Shader）和像素着色器（Fragment Shader）。

顶点着色器输入的是顶点相关的数据，如位置、法向、颜色等。顶点着色器里面能够实现下面的功能：使用模型视图矩阵及投影矩阵进行顶点变换；法线变换及归一化；纹理坐标生成和变换；逐顶点或逐像素光照计算；颜色计算。每个顶点着色器不一定必须实现上面所有操作，如可以不使用光照。但假如使用了顶点着色器，顶点处理器的所有 OpenGL 提供的固定功能都将被替换。因此不能在 GLSL 里面只编写法线变换部分，而采用 OpenGL 内置的固定功能完成纹理坐标生成。顶点处理器并不知道连接信息，因此不能执行拓扑信息有关的操作。比如顶点处理器不能进行背面剔除，它只是操作顶点而不是面。顶点着色器至少需要一个变量 gl_Position，通常要用模型视图矩阵及投影矩阵进行变换。顶点处理器可以访问 OpenGL 状态，所以可以用来处理材质和光照。

像素着色器能实现如下功能：逐像素计算颜色和纹理坐标；应用纹理；雾化计算；如果需要逐像素光照，可以用来计算法线；像素着色器的输入来自于顶点着色器的输出。输入的顶点坐标、颜色、法线都是经过顶点着色器输出经过插值的结果。因为顶点着色器是在顶点上计算的，而像素着色器是在像素上计算的，顶点变换后的位置并不一定能够覆盖所有像素，因此每个像素上的值需要通过顶点变换后的位置来进行插值。使用 GLSL 编写像素着色器后，OpenGL 本身提供的固定函数都被替换了，所有和像素相关的处理，假如需要用到的话，都需要用 GLSL 编写，而不能默认为某些用 GLSL 编写，其他的还使用原来 OpenGL 内置的功能。例如，无法只编写贴图 GLSL 程序，而同时采用 OpenGL 内置的雾化功能。像素着色器的输出就是每个像素的最终颜色，也就是给 gl_FragColor 赋值。

3.4 数据传递

GLSL 程序的数据是 OpenGL 传递的。这种数据的传递是单向的，也就是 OpenGL 把数据传递给 GLSL，而 GLSL 的数据输出给缓存。OpenGL 无法得到 GLSL 输出的数据。OpenGL 传递给 GLSL 数据的方式有以下几种。

（1）OpenGL状态

（2）Uniform变量

（3）Attribute变量

（4）贴图

1. OpenGL状态

第1种是通过状态传递，GLSL可以读取OpenGL的状态，因此通过设置OpenGL的状态就可以把数据传递给GLSL。例如，在OpenGL中设置光的颜色，这是一个状态，GLSL就可以得到这个数据。

通过Uniform和Attribute变量传递的数据在GLSL中是只读的，不能修改。Uniform变量可以定义在顶点着色器和像素着色器中。Attribute变量只能定义在顶点着色器中。Attribute变量是定义在每个顶点上，也就是Attribute变量的值每个顶点都不同，如顶点的位置、法向。Uniform变量不随顶点变化，如时间等。Uniform变量不能在OpenGL的函数GL.Begin和GL.End函数之间设置。

2. Uniform变量

OpenGL需要如下两个函数进行变量传递。

（1）获取变量的位置

```
GL.GetUniformLocation
```

（2）传递变量

```
GL.Uniform1
```

实例如下。

（1）得到GLSL变量内存位置的实例如下。

```
GL.GetUniformLocaton(this.handle,"FrameWidth");
```

其中FrameWidth就是在GLSL程序里定义的变量名字，OpenGL在此处必须保持和GLSL定义的变量名一样。假如GLSL程序里面修改了这个变量名，在OpenGL在传递数据时，也需要修改这个变量名。

（2）传递一个float类型的数据如下。

```
GL.Uniform1(GL.GetUniformLocation(this.handle,"FrameWidth"),
                                    ConfigGPUHeat.Instance.FrameWidth);
```

（3）可以传递的变量类型是浮点数、整数、布尔值、向量、矩阵、纹理贴图等。传递变量的函数根据变量类型里包含元素数量不同的不同，做相应变化，示例如下。

```
GL.Uniform1;
GL.Uniform2;
GL.Uniform3;
GL.Uniform4;
GL.UniformMatrix2;
GL.UniformMatrix2x3;
GL.UniformMatrix2x4;
```

```
GL. UniformMatrix3;
GL. UniformMatrix3x2;
GL. UniformMatrix3x4;
GL. UniformMatrix4;
GL. UniformMatrix4x2;
GL. UniformMatrix4x3;
```

（4）以下是传递某个 GLSL 程序变量的函数实例。

```
public override void PassParameter( )
        {
            GL. Uniform1( GL. GetUniformLocation( this. handle, "FrameWidth" ),
                        ConfigGPUHeat. Instance. FrameWidth);
            GL. Uniform1( GL. GetUniformLocation( this. handle, "FrameHeight" ),
                        ConfigGPUHeat. Instance. FrameHeight);
            GL. Uniform1( GL. GetUniformLocation( this. handle, "Frequency" ),
                        ConfigGPUHeat. Instance. Frequency);
            GL. Uniform1( GL. GetUniformLocation( this. handle, "Speed" ),
                        ConfigGPUHeat. Instance. Speed);
            GL. Uniform1( GL. GetUniformLocation( this. handle, "Fade" ),
                        ConfigGPUHeat. Instance. Fade);
            GL. Uniform1( GL. GetUniformLocation( this. handle, "Offset" ),
                        ConfigGPUHeat. Instance. Offset);
            GL. Uniform1( GL. GetUniformLocation( this. handle, "FrameBuffer" ),
                        OpenGLManager. Instance. FirstTexture);

        }
```

3. Attribute 变量

Uniform 传递的变量不能针对每个顶点，如果需要在顶点上设置一些数据，就需要 Attribute 变量。Attribute 变量数据是针对顶点的，因此只能在顶点着色器中定义和使用，而不能用在像素着色器中定义和使用。和 Uniform 变量的传递方法类似，也是分为两步。

首先通过得到变量在内存中的位置。

```
GL. GetAttribLocation( this. handle, name);
```

然后把变量传递过去：

```
GL. VertexAttrib1;
```

根据要传递的变量类型和数量不同，调用不同的传递函数。

```
GL. VertexAttrib1;
GL. VertexAttrib2;
GL. VertexAttrib3;
GL. VertexAttrib4;
```

```
GL. VertexAttrib4N;
GL. VertexAttribI1;
GL. VertexAttribI2;
GL. VertexAttribI3;
GL. VertexAttribI4;
GL. VertexAttribIPointer;
GL. VertexAttribIPointer < > ;
GL. VertexAttribPointer;
GL. VertexAttribPointer < > ;
```

Attribute 变量的传递可以放在 GL. Begin 和 GL. End 中，代码如下。

```
protected override void RenderShape( )
    {
        GL. Color3(1f,1f,1f);
        GL. Begin(BeginMode. Quads);
        {
            GL. Normal3(pl. Normal);
            GL. VertexAttrib3(AttribTangent,ref Tangent);
            GL. MultiTexCoord2(TextureUnit. Texture0,0f,1f);
            GL. Vertex3( -1. 0f,1. 0f,0. 0f);

            GL. Normal3(pl. Normal);
            GL. VertexAttrib3(AttribTangent,ref Tangent);
            GL. MultiTexCoord2(TextureUnit. Texture0,0f,0f);
            GL. Vertex3( -1. 0f, -1. 0f,0. 0f);

            GL. Normal3(pl. Normal);
            GL. VertexAttrib3(AttribTangent,ref Tangent);
            GL. MultiTexCoord2(TextureUnit. Texture0,1f,0f);
            GL. Vertex3(1. 0f, -1. 0f,0. 0f);

            GL. Normal3(pl. Normal);
            GL. VertexAttrib3(AttribTangent,ref Tangent);
            GL. MultiTexCoord2(TextureUnit. Texture0,1f,1f);
            GL. Vertex3(1. 0f,1. 0f,0. 0f);
        }
        GL. End( );
    }
```

4. 贴图

贴图也可以传递数据，图片本身可以包含像素信息，也可以包含其他信息。也就是可以把图片里像素的值解释为任何自定义的数据。根据应用的不同，可以作为向量、矩阵、速度等信息传递。OpenGL 的状态信息也可以解释为其他自定义的信息，例如光照颜色的 OpenGL

状态可以解释为变化的速率等。因为传递进来的是个数值。这个数值的变量名和变量类型本身并没有具体的含义，无论是贴图还是 OpenGL 状态，都不能限制所包含数值的具体使用。因此，可以根据需要把这些数值解释为相应的数据。

5. Varying 变量

上面两种变量都是把数据从 OpenGL 传递到 GLSL。在 GLSL 内部有两个着色器，GLSL 程序先执行顶点着色器，然后再执行像素着色器。Varying 变量就是把顶点着色器的数据输出传递到像素着色器中。Varying 变量必须在两个着色器中都定义为一样的名字才能传递。两个一样名字的 Varying 数据值不一样，是经过自动插值的。顶点着色器输出的 Varying 数据在传递过程中自动插值得到像素着色器中的 Varying 数据。代码如下。

```
varying vec3 normal;
varying vec3 vertex;
```

第 4 章 渲染光照

4.1 没有光照

最简单的光照模式就是没有光照，只有给三维模型设置颜色。最简单的 GLSL 程序在顶点着色器里只进行最基本的变换，不做其他任何操作。在像素着色器里只设置输出的颜色。每个着色器都只有一行代码。这个最简单的程序没有考虑的光照，只把三维的顶点通过变换矩阵变换到二维，然后上色。因此渲染出来的效果没有三维感觉，也就是没有明暗效果。通过这个最简单的 GLSL 程序可以了解 GLSL 程序编写的模式。因为 GLSL 程序分为顶点着色器和像素着色器两种，从而即使是最简单的程序也需要分为这两部分来分别编写，这是由 GPU 编程的架构所决定的。下面是相应的 GLSL 着色器代码。

1. 顶点着色器

有 3 种等价的方式。

(1)

```
void main(void)
  {
   gl_Position = gl_ProjectionMatrix * gl_ModelViewMatrix * gl_Vertex;
  }
```

(2)

```
void main(void)
  {
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
  }
```

(3)

```
void main(void)
  {
    gl_Position = ftransform();
  }
```

也可以把 3 种方法放到一个程序里面，根据参数进行选择。

```
uniform int type;
```

```
void main(void)
{
    if(type == 1)
    {
        gl_Position = ftransform();
    }
    else if(type == 2)
    {
        gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    }
    else if(type == 3)
    {
        gl_Position = gl_ProjectionMatrix * gl_ModelViewMatrix * gl_Vertex;
    }
}
```

在上述程序中，其中：

gl_ProjectionMatrix 表示投影矩阵；

gl_ModelViewMatrix 表示模型视图变换矩阵；

gl_Vertex 表示顶点的坐标；

gl_Position 是经过变换后的位置；

gl_ModelViewProjectionMatrix 表示投影矩阵和模型视图矩阵的乘积。

这几个变量都是 GLSL 内置变量，可以直接调用。这些内置变量的值都是在 OpenGL 里面赋值的。采用矩阵乘积矩阵不需要每次都进行两个矩阵相乘，从而加快渲染速度。这个最简单的操作是每个 GLSL 函数都必须进行的，因此 GLSL 使用 ftransform() 函数是针对硬件进行优化的，速度更快。

2. 像素着色器

在像素着色器代码里 gl_FragColor 是内置的颜色变量，通过给这个变量赋值就得到模型最终的输出颜色。在最简单的渲染里面，颜色可以内置在 GLSL 程序中，也可以通过 OpenGL 外部传递颜色进来。除了可以使用 Uniform 变量进行传递以外，还可以使用 OpenGL 的状态进行传递。

```
uniform vec3  Color;
 void main()
{
    gl_FragColor = vec4(Color,1.0);
}
```

3. 参数设置代码

```
public class ConfigGPUSimplest:ConfigGPUBase
    {
        private static ConfigGPUSimplest singleton = new ConfigGPUSimplest();
```

```
        public static ConfigGPUSimplest Instance
        {
            get
            {
                if (singleton == null)
                    singleton = new ConfigGPUSimplest();
                return singleton;
            }
        }

        private int type = 1;
        public int Type
        {
            get
            {
                return type;
            }
            set
            {
                type = value;
            }
        }
        private Color color = Color.FromArgb(255,0,191,191);
        public Color Color
        {
            get
            {
                return color;
            }
            set
            {
                color = value;
            }
        }
    }
```

4. 外部传递参数程序

这个程序用来设定 GLSL 程序的文件位置，以及传递参数到 GLSL 程序内部。

```
    public class GPURenderSimplest : AbstractGPURender
    {
        public GPURenderSimplest()
        {
```

```
            VSFileName = "GLSL/Simplest.vert";
            FSFileName = "GLSL/Simplest.frag";
        }
        public override void PassParameter()
        {
            GL.Uniform1(GL.GetUniformLocation(this.handle,"type"),
                        ConfigGPUSimplest.Instance.Type);

            GL.Uniform3(GL.GetUniformLocation(this.handle,"Color"),
                    GetColor(ConfigGPUSimplest.Instance.Color));
        }
    }
```

5. 效果图

图 4-1 所示是几个三维模型用最简单的 GLSL 程序进行渲染得到的效果图。可以看出，由于没有考虑光照，三维模型每个顶点无法显示，这是因为光源位置和方向不同而产生的明暗效果，从而最终只显示出简单的平面效果。

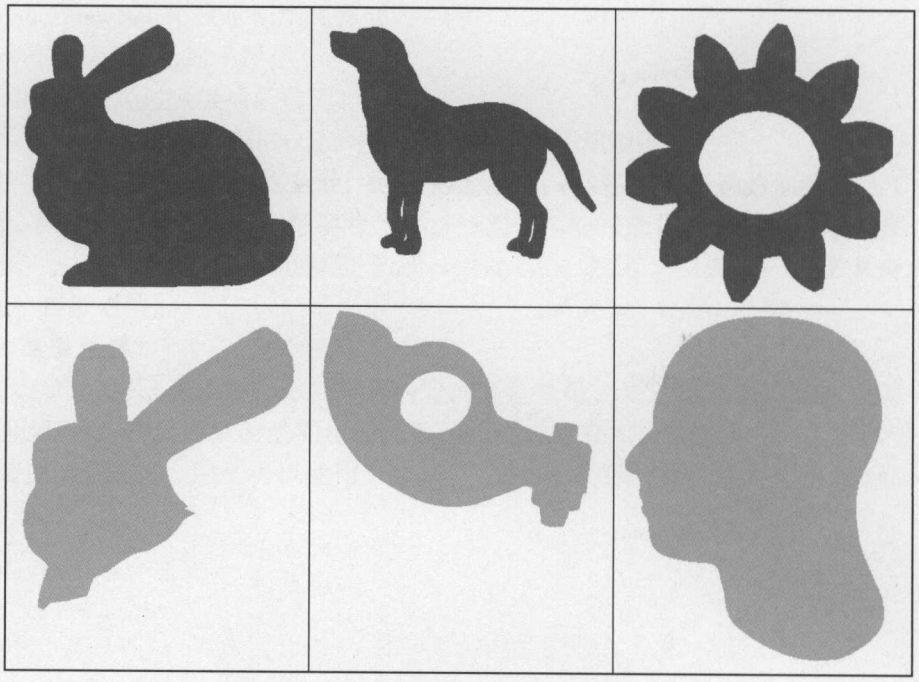


图 4-1　最简单 GLSL 程序渲染效果

在 OpenGL 和 GLSL 的交互中，最重要的一部分是 OpenGL 内置状态变量到 GLSL 程序的传递，也就是 GLSL 程序可以得到 OpenGL 设置的内置变量，步骤如下。

第一步：OpenGL 程序通过 GL.Color 函数设置颜色状态信息。

第二步：顶点着色器通过内置变量 gl_Color 接收颜色值。

第三步：顶点着色器计算正面和反面的颜色，然后分别保存在 gl_FrontColor 和 gl_BackColor 变量中。

第四步：顶点着色器中的内置变量 gl_FrontColor 和 gl_BackColor 通过自动插值产生像素着色器中的内置变量 gl_Color 的值。

第五步：像素着色器通过内置的 Varying 变量 gl_Color 设置 gl_FragColor 的值。

一般来说，顶点着色器和像素着色器传递的 Varying 变量的名字要一样，gl_FrontColor 和 gl_BackColor 对应到 gl_Color 是个特例。同时，像素着色器里的 gl_Color 变量和顶点着色器里的 gl_Color 名字虽然一样，但并不冲突。

4.2　扁平渲染

扁平渲染比 4.1 节的渲染效果复杂一点。扁平渲染是把三维的模型的顶点坐标 Z 设置为 0 得到的渲染效果，从而使三维模型丧失了三维立体感觉，呈现为平面的感觉。扁平渲染中三维模型在矩阵变换之前就映射到平面上，已经变为二维模型，然后再对二维模型进行各种矩阵变换。这是一种利用 GLSL 程序改变三维模型形状来达到某种特殊渲染效果的方法。

1. 代码

顶点着色器的代码如下。

```
void main(void)
{
    vec4 v = vec4(gl_Vertex);
    v.z = 0.0;
    gl_Position = gl_ModelViewProjectionMatrix * v;
}
```

还可以对 Z 坐标进行其他变换，如正弦或余弦变换，得到波浪效果。除了 Z 坐标之外，还可以对 X，Y 坐标做各种需要的变换来达到其他的渲染效果。

```
void main(void)
{
    vec4 v = vec4(gl_Vertex);
    v.z = sin(16.0 * v.x) * 0.35;
    gl_Position = gl_ModelViewProjectionMatrix * v;
}
```

像素着色器代码和最简单的无光照渲染代码一样。

2. 效果图

如图 4-2 所示的第 1 行中对三维兔子模型进行了扁平渲染，从中可以看出三维模型不管从任何角度看都显示出平面的感觉。第 2 行显示的是进行正弦、余弦变换的波浪效果。

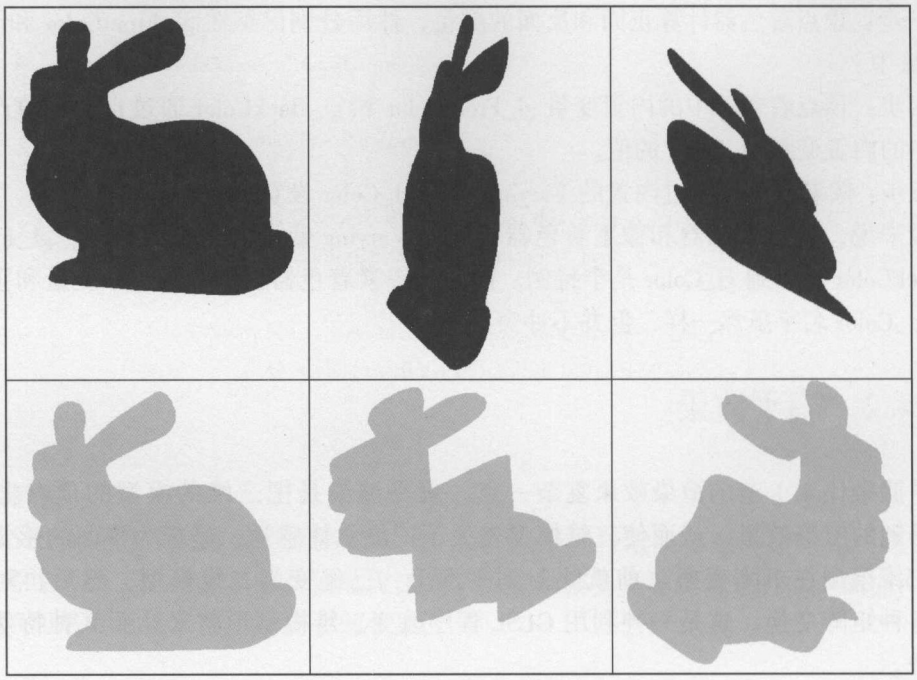


图 4-2　扁平渲染效果

4.3　最简单光照

前几节的渲染效果都是没有光照的，也就是不需要设置光源的位置和法向。如果没有光源，三维模型显示出来就是平面的。计算光照最重要部分是计算三维模型每个顶点的法向，有了顶点的法向才能够判断当前顶点和光源的相对位置和相对方向，从而得到光源在此顶点上最终生成的颜色值。在 OpenGL 和 GLSL 里做的光照和真实的物理光照不一样，是对真实光照的近似。因此，可以有各种各样的近似计算方法。下面是一种最简单的近似光照算法 GLSL 程序。

1. 顶点着色器

```
varying vec3 Normal;
Void main(void)
{
gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
Normal = normalize(gl_NormalMatrix * gl_Normal);
gl_FrontColor = gl_Color;
}
```

其中：

gl_Normal 是 GLSL 内置变量，传递的是顶点的原始法向，也就是没有变换过的法向；

gl_NormalMatrix 是法向的变换矩阵；

Normal 是 Varying 变量，用来把变换后的法向传递到像素着色器中。

这是因为 gl_Normal 等顶点上的数据 OpenGL 只能传递给顶点着色器，而无法从 OpenGL 传递给像素着色器。因此需要额外定义一个 Varying 变量，来传递法向数据。

2. 法向变换矩阵

每个顶点在着色时需要把顶点位置变换到视点空间，也就是通过模型视点矩阵进行变换。那么相应的每个顶点的法向也需要进行变换。

顶点变换的方法如下。

```
gl_ModelViewMatrix * gl_Vertex
```

假如变换只有旋转和平移，而没有缩放时，法向变换也可以用如下方法。

```
Vec3(gl_ModelViewMatrix * vec4(gl_Normal,0.0));
```

但如果缩放是一致的话，法向只是大小发生变化，法向不变；如果缩放是非一致的话，那么法向的方向就会发生变化。法向变换如图 4-3 所示。

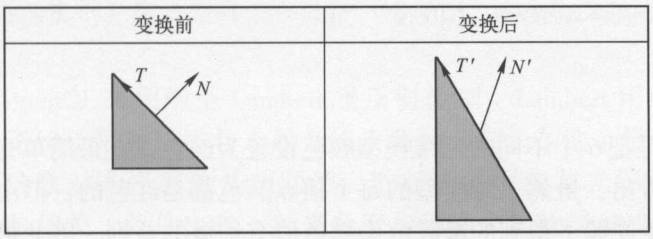


图 4-3　法向变换

因此模型视图矩阵不能直接用来进行法向变换。法向变换矩阵的推导如下。

第一步：变换前切线和法向垂直，也就是

$$\boldsymbol{T} \cdot \boldsymbol{N} = 0$$

第二步：变换后依然垂直，也就是

$$\boldsymbol{T}' \cdot \boldsymbol{N}' = 0$$

第三步：用 $\boldsymbol{G}$ 表示法向变换矩阵，用 $\boldsymbol{M}$ 表示模型视图左上 3×3 的矩阵，那么

$$\boldsymbol{N}' \cdot \boldsymbol{T}' = (\boldsymbol{GN}) \cdot (\boldsymbol{MT}) = 0$$

第四步：从而得到

$$(\boldsymbol{GN}) \cdot (\boldsymbol{MT}) = (\boldsymbol{GN})^{\mathrm{T}}(\boldsymbol{MT})$$

第五步：继续置换可以得到

$$(\boldsymbol{GN})^{\mathrm{T}}(\boldsymbol{MT}) = \boldsymbol{N}^{\mathrm{T}}\boldsymbol{G}^{\mathrm{T}}\boldsymbol{MT}$$

第六步：从上面的公式可以知道，因为法向和切线点积为 0，因此

$$\boldsymbol{G}^{\mathrm{T}}\boldsymbol{M} = \boldsymbol{I}$$

第七步：最终得出法向变换矩阵为

$$\boldsymbol{G} = (\boldsymbol{M}^{-1})^{\mathrm{T}}$$

第八步：假如 $\boldsymbol{M}$ 是正交矩阵，可以得出

$$\boldsymbol{M}^{-1} = \boldsymbol{M}^{\mathrm{T}} \Rightarrow \boldsymbol{G} = \boldsymbol{M}$$

第九步：也就是法向变换矩阵使模型视图矩阵左上 3×3 矩阵的逆矩阵转置。这个矩阵

被 OpenGL 保存在 gl_ NormalMatrix 变量里面。

第十步：法向在变换后需要重新归一化，归一化的法向在从顶点着色器传递到像素着色器中时因为插值的原因需要再一次进行归一化。

3. 像素着色器

像素着色器通过内置的光源位置、光源颜色和变换后的法向得到最终的显示颜色。也可以把光源位置、方向、颜色设置为 Uniform 变量，从而可以从外部更改这些参数。

```
varying vec3 Normal;
vec3 lightColor = vec3(1.0,0.0,0.0);
vec3 lightDir = vec3(0.0,0.0,4.0);
void main(void)
{
    vec3 color = clamp(dot(normalize(Normal),lightDir),
                       0.0,1.0) * lightColor;
    gl_FragColor = vec4(color,1.0);
}
```

4. 效果图

如图4-4 所示是把6 个不同的三维模型颜色设置为红色，在最简单光照效果下得到的渲染结果。从中可以看出，虽然三维模型的每个顶点颜色都是红色的，但由于每个顶点法向不一样，从而在同一个光源下得到的最终渲染颜色值有细微的差别。因此最终的模型呈现立体效果。

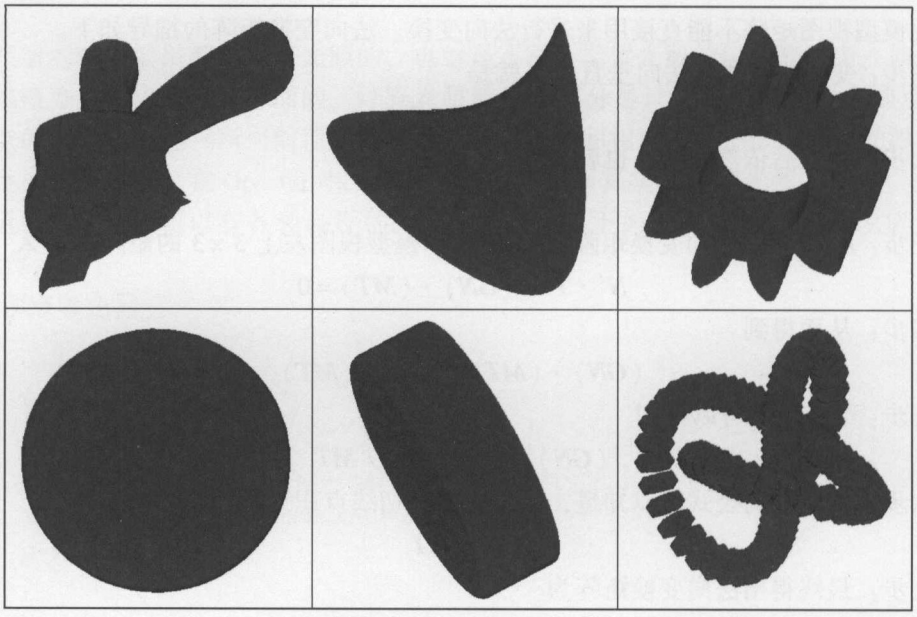


图 4-4　最简单光照效果

4.4　逐顶点光照

4.4.1　光照模型

三维渲染中的光照需要根据光照模型来进行计算。光照模型就是用来近似真实物理世界光照的方法。有很多光照模型，OpenGL 中内置了一种最常用的光照模型。这种光照模型也可以在 GLSL 中实现，从而可以更好地了解 OpenGL 光照模型的计算原理和方法。基于这种基本的光照模型也可以开发更多的其他光照模型。这种光照模型和真实光照相比，由于采用了简化的方法，因此计算量少、速度快。光照模型计算可以在顶点着色器中完成，称为逐顶点光照；也可以在像素着色器中完成，称为逐像素光照。

OpenGL 中的光照模型把光分为 3 种类型光源：方向光（Directional）、点光（Point）、聚光（Spotlight）。每种类型光源的光有 3 个分量：漫反射、高光和环境光。

1. 漫反射

对于漫反射，OpenGL 采用的是 Lambert 漫反射模型。Lambert 用余弦定律描述了平面散射光的亮度，正比于平面法线与入射光线夹角的余弦。在这个模型中，不管观察者的角度如何，得到的散射光强度总是相同的。散射光的强度与光源中散射光成分及材质中散射光反射系数相关，此外也和入射光角度与物体表面法线的夹角相关，如图 4-5 所示。

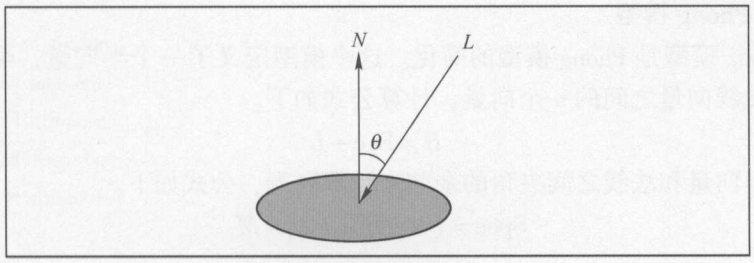


图 4-5　漫反射

Lambert 漫反射模型得到的漫反射光计算公式如下。

$$I_a = L_d * M_d * \cos(\theta)$$

式中，I_a是反射光的强度；L_d是光源的散射成分；M_d是材质的散射系数。

在 OpenGL 固定渲染内置的函数中，已经实现了这个漫反射模型公式计算。只要设置相关的外部光源、材质的参数就可以了。但如果用 GLSL，则需要编写代码来计算 Lambert 公式。

2. 环境光

根据全局的环境光参数和材质、光源的环境光系数，可以计算环境光，公式如下。

$$I_a = G_a * M_a + L_a * M_a$$

3. 高光

高光也是镜面反射光，也就是三维模型光滑的部位在光源的照射下看起来很明亮的效果。在 OpenGL 中，高光采用计算模型称为 Phong 模型，镜面反射成分和反射光线与视线夹角的余弦相关，如图 4-6 所示。

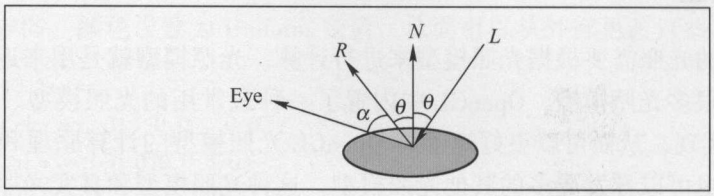


图 4-6　高光反射

图中，L 表示入射光；N 表示法线；Eye 表示从顶点指向观察点的视线；R 是 L 经镜面反射后的结果。

镜面反射成分与 α 角的余弦相关。

假如视线正好和反射光重合，则接收到最大的反射强度。当视线与反射光相分离时，反射强度将随之下降，下降速率可以由名为 shininess 的因子控制，shininess 的值越大，下降速率越快。也就是说，shininess 越大的话，镜面反射产生的亮点就越小。在 OpenGL 中这个因子的值范围是 0 ~ 128。

从而计算公式为：

$$R = -2N(L \cdot N) + L$$

4. Blinn - Phong 模型

Blinn - Phong 模型是 Phong 模型的简化。这个模型定义了一个半向量，半向量是方向处在观察向量及光线向量之间的一个向量，计算公式如下。

$$H = \text{Eye} - L$$

可以利用半向量和法线之间夹角的余弦来计算高光，公式如下。

$$\text{Spec} = (N \cdot H)^{s} * L_s * M_s$$

4.4.2　参数和步骤

OpenGL 和 GLSL 的数据传递可以通过 Uniform 变量来进行，光源、材质等相关参数是内置的 Uniform 变量，不需要重新定义。这些参数只需在 OpenGL 中进行设置，就可以直接在 GLSL 中使用。这些参数不需要在 GLSL 代码中进行声明，GLSL 已经内置了定义声明。其实也可以自定义类似的结构体和变量，但需要手工来传递这些参数，而不能使用 OpenGL 的内置参数。

1. 光源参数

```
struct gl_LightSourceParameters
{
    vec4 ambient;
```

```
    vec4 diffuse;
    vec4 specular;
    vec4 position;
    vec4 halfVector;
    vec3 spotDirection;
    float spotExponent;
    float spotCutoff;
    float spotCosCutoff;
    float constanctAttenuation;
    float linearAttenuation;
    float quadraticAttenuation;

};
```

2. 光源模式

```
struct gl_LightModelParameters
{
    vec4 ambient;
};
```

3. 材质参数

```
struct gl_MaterialParameters
{
    vec4 emission;
    vec4 ambient;
    vec4 diffuse;
    vec4 specular;
    float shininess;
};
```

4. 参数定义

```
uniform gl_LightSourceParameters gl_LightSource[gl_MaxLights];
uniform gl_LightModelParameters gl_LightModel;
uniform gl_MaterialParameters gl_FrontMaterial;
uniform gl_MaterialParameters gl_BackMaterial;
```

5. 第1个光源的散射成分

```
gl_LightSource[0].diffuse
```

6. 材质的散射系数

```
gl_FrontMaterial.diffuse
```

7. 光源的方向

在方向光中，OpenGL 的光源方向是指向光源的。

```
gl_LightSource[0].position
```

8. 半向量

半向量可以从 OpenGL 直接得到，不需要计算。

```
gl_LightSource[0].halfVector
```

计算步骤如下。

第一步：用矩阵对法向进行变换。

```
normal = normalize(gl_NormalMatrix * gl_Normal);
```

第二步：对光源方向进行归一化。

```
lightDir = normalize(vec3(gl_LightSource[0].position));
```

第三步：计算光源法向和每个顶点法向的夹角余弦。

```
NdotL = max(dot(normal,lightDir),0.0);
```

第四步：用材质漫反射系数和光源漫反射参数，计算漫反射的分量。

```
diffuse = gl_FrontMaterial.diffuse * gl_LightSource[0].diffuse;
```

第五步：计算环境光。

```
ambient = gl_FrontMaterial.ambient * gl_LightSource[0].ambient;
```

第六步：计算高光。

```
NdotHV = max(dot(normal,gl_LightSource[0].halfVector.xyz),0.0);
specular = gl_FrontMaterial.specular * gl_LightSource[0].specular *
        pow(NdotHV,gl_FrontMaterial.shininess);
```

第七步：计算总的光照。

```
gl_FrontColor = NdotL * diffuse + globalAmbient + ambient;
```

4.4.3 代码和效果

下面是完整的光照模型 GLSL 计算代码，和其他渲染效果一样，分为顶点着色器代码和像素着色器代码。其中光照模型的计算部分全部放在顶点着色器代码中。

1. 顶点着色器

```
void main()
{
    vec3 normal,lightDir;
    vec4 diffuse,ambient,specular,globalAmbient;
    float NdotL,NdotHV;
```

```
    normal = normalize( gl_NormalMatrix * gl_Normal) ;
    lightDir = normalize( vec3( gl_LightSource[0]. position)) ;
    NdotL = max( dot( normal, lightDir) ,0. 0) ;
    if ( NdotL > 0. 0)
    {
    NdotHV = max( dot( normal, gl_LightSource[0]. halfVector. xyz) ,0. 0) ;
    specular = gl_FrontMaterial. specular * gl_LightSource[0]. specular *
            pow( NdotHV, gl_FrontMaterial. shininess) ;
    }
    diffuse = gl_FrontMaterial. diffuse * gl_LightSource[0]. diffuse;
    ambient = gl_FrontMaterial. ambient * gl_LightSource[0]. ambient;
    globalAmbient = gl_FrontMaterial. ambient * gl_LightModel. ambient;
    gl_FrontColor = NdotL * diffuse + globalAmbient + ambient;
    gl_Position = ftransform( ) ;
}
```

2. 像素着色器

因为光照计算都是在顶点着色器中进行的，像素着色器只需赋值颜色。

```
void main( )
{
    gl_FragColor = gl_Color;
}
```

3. GLSL 参数传递程序

由于这个 GLSL 光照程序中用到的光源颜色、方向灯参数都是通过 OpenGL 内置状态来传递的，因此不需要通过 Uniform 变量来传递参数。整个代码比较简单，只要设置相应的 GLSL 程序文件路径即可。

```
public class GPURenderLightPerVertex : AbstractGPURender
    {
        public GPURenderLightPerVertex( )
        {
            VSFileName = "GLSL/LightingPerVertex. vert" ;
            FSFileName = "GLSL/LightingPerVertex. frag" ;
        }
    }
```

4. 效果图

如图4-7所示的第一行分别把光源的颜色设置为不同的3种颜色，第二行保持光源白色不变，把材质的颜色设置为3种不同的颜色。可以看出，这种 OpenGL 内置的光照模型和前面的光照相比已经可以得到更加真实的渲染效果。

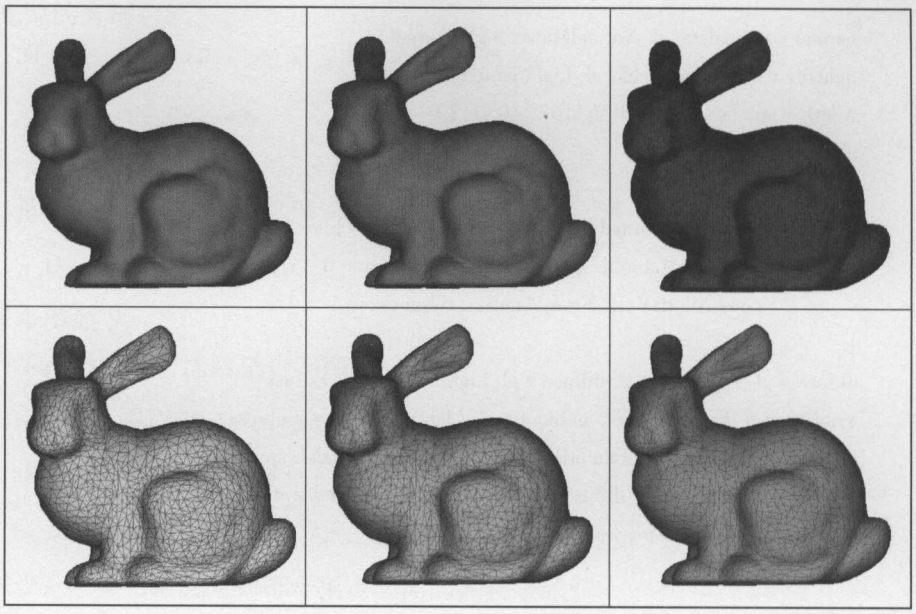


图 4-7　逐顶点光照效果

4.5　逐像素光照

在光照模型中，每个顶点接收到的信息有法线、半向量、光源方向等。其中法线需要变换到视点空间然后归一化，半向量和光源方向也需要归一化。在 4.4 节逐顶点光照中，这些向量直接在顶点着色器中计算出最后的光照结果，也就是每个顶点上的颜色，这些颜色再进行插值传递给每个像素。而在逐像素光照中，这些向量被传递给像素着色器，这些归一化之后的向量在从顶点着色器传递到像素着色器中时会自动进行插值。然后在像素着色器中的每个像素上，再根据光照模型的公式计算出每个像素上的最终颜色。因为在像素着色器中是作用在每个像素上，因此比作用在每个顶点上更精确。逐像素光照技术中光照公式和算法与逐顶点光照技术一样，只是把顶点着色器中的部分功能放到像素着色器中来实现。

1. 顶点着色器

顶点着色器中不直接计算最终每个顶点的光照效果，而是计算漫反射、环境光、光源方向、顶点法向等中间值，这些中间值通过 varying 变量传递给像素着色器。

```
varying vec4 diffuse,ambient;
varying vec3 normal,lightDir,halfVector;
void main()
{
    normal = normalize(gl_NormalMatrix * gl_Normal);
    lightDir = normalize(vec3(gl_LightSource[0].position));
    halfVector = normalize(gl_LightSource[0].halfVector.xyz);
```

```
    diffuse = gl_FrontMaterial. diffuse * gl_LightSource[0]. diffuse;
    ambient = gl_FrontMaterial. ambient * gl_LightSource[0]. ambient;
    ambient  + = gl_FrontMaterial. ambient * gl_LightModel. ambient;
    gl_Position = ftransform( );
}
```

2. 像素着色器

在像素着色器中，通过 varying 变量传递进来的定义在每个顶点上的中间值会自动被插值到每个像素上，然后再使用光照模型公式，在每个像素上计算光照的最终渲染颜色值。

```
varying vec4 diffuse,ambient;
varying vec3 normal,lightDir,halfVector;
void main( )
{
  vec3 n,halfV;
  float NdotL,NdotHV;
  vec4 color = ambient;
  n = normalize( normal) ;
  NdotL = max( dot( n,lightDir) ,0. 0) ;
  if ( NdotL  > 0. 0)
  {
  color  + = diffuse * NdotL;
  halfV = normalize( halfVector) ;
  NdotHV = max( dot( n,halfV) ,0. 0) ;
  color  + = gl_FrontMaterial. specular  *
             gl_LightSource[0]. specular  *
             pow( NdotHV,gl_FrontMaterial. shininess) ;
  }
  gl_FragColor = color;
}
```

3. GLSL 参数传递程序

```
public class GPURenderLightPerFrag : AbstractGPURender
    {
        public GPURenderLightPerFrag( )
        {
            VSFileName = " GLSL/LightingPerFrag. vert" ;
            FSFileName = " GLSL/LightingPerFrag. frag" ;
        }
    }
```

4. 光源参数设置

由于使用的是 OpenGL 内置光源来传递光源的参数，因此需要在 OpenGL 代码里设置光源参数。

```
GL. Enable( EnableCap. Light0) ;
Vector4 pos =
    new Vector4( ( float) GlobalSetting. Light0Setting. LightPosition. x,
                 ( float) GlobalSetting. Light0Setting. LightPosition. y,
                 ( float) GlobalSetting. Light0Setting. LightPosition. z,
                 0. 0f) ;

GL. Light( LightName. Light0, LightParameter. Position, pos) ;

GL. Light( LightName. Light0, LightParameter. Diffuse,
          GlobalSetting. Light0Setting. DiffuseColor) ;
GL. Light( LightName. Light0, LightParameter. Specular,
          GlobalSetting. Light0Setting. SpecularColor) ;
GL. Light( LightName. Light0, LightParameter. Ambient,
          GlobalSetting. Light0Setting. AmbientColor) ;
```

5. 效果图

如图 4-8 所示是几个不同的三维模型在不同光照颜色下的逐像素光照效果。

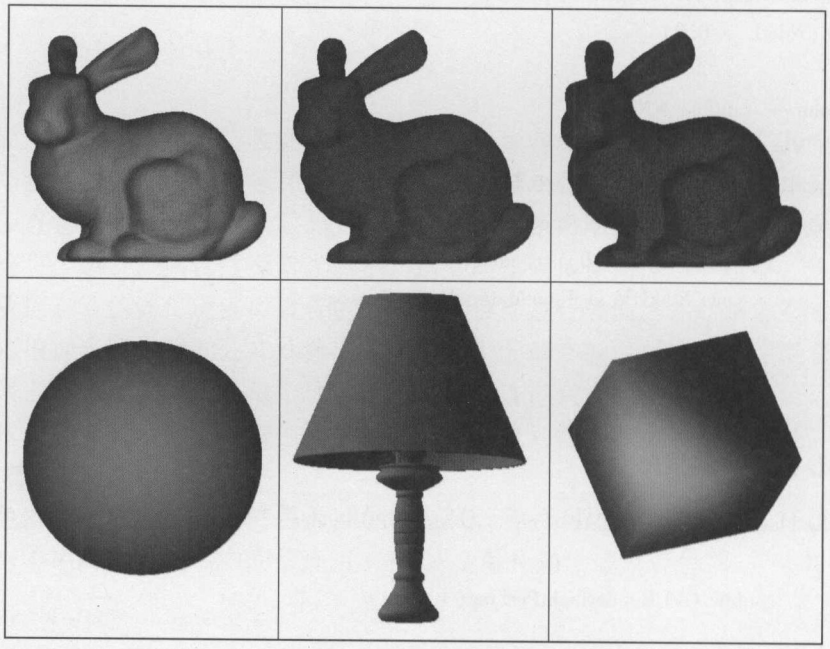


图 4-8　逐像素光照效果

4.6　其他光源类型

4.6.1　点光源

前面的光照模型中使用的光是方向光。方向光假设光源在无限远的地方，到达物体

时是平行光。除了方向光光源外，还有点光源和聚光灯光源。点光源有一个空间中的位置，并向各个方向辐射光线。点光源的强度会随到达三维模型的距离而衰减。点光源的衰减由3个系数决定：一个常数项、一个线性项和一个二次项。在OpenGL中衰减计算公式如下。

$$\text{att} = \frac{1}{k_0 + k_1 d + k_2 d^2}$$

式中，k_0是常数衰减系数；k_1是线性衰减系数；k_2是二次衰减系数；d是光源位置到顶点的距离。

衰减与距离是非线性关系，因此不能逐顶点计算衰减再在像素着色器中使用插值结果，但可以在顶点着色器中计算距离，然后把距离插值到像素着色器中计算衰减。

最终的点光源光照公式为

$$\text{color} = \text{ambientGloabl} + \text{att}(\text{dmbient} + \text{diffuse} + \text{specular})$$

其中环境光部分分解为两项：使用光照模型的全局环境光设置和光源中的环境光设置。两个环境光分别计算。在OpenGL中光源的position域的w分量：对方向光来说它是0，对点光来说，这个分量是1。对方向光来说，光线的方向对所有顶点相同，但对点光来说，方向是从顶点指向光源位置的向量。因此在顶点着色器中需要计算每个顶点的光照方向。

1. 顶点着色器

顶点着色器代码和方向光计算的代码大部分相同，不同的地方是需要根据每个顶点的位置和光源的位置来计算每个顶点的光照方向，代码如下。

```
ecPos = gl_ModelViewMatrix * gl_Vertex;
aux = vec3(gl_LightSource[0].position - ecPos);
lightDir = normalize(aux);
```

顶点着色器完整代码如下。

```
varying vec4 diffuse,ambientGlobal,ambient;
varying vec3 normal,lightDir,halfVector;
varying float dist;
void main()
{
    vec4 ecPos;
    vec3 aux;
    normal = normalize(gl_NormalMatrix * gl_Normal);
    ecPos = gl_ModelViewMatrix * gl_Vertex;
    aux = vec3(gl_LightSource[0].position - ecPos);
    lightDir = normalize(aux);
    dist = length(aux);
    halfVector = normalize(gl_LightSource[0].halfVector.xyz);
    diffuse = gl_FrontMaterial.diffuse * gl_LightSource[0].diffuse;
    ambient = gl_FrontMaterial.ambient * gl_LightSource[0].ambient;
```

```
    ambientGlobal = gl_FrontMaterial. ambient * gl_LightModel. ambient;
    gl_Position = ftransform( );
}
```

2. 像素着色器

像素着色器代码根据点光源的光照公式计算每个像素上在点光源的照射下的漫反射、环境光等颜色值，然后得到每个像素最终的颜色。

```
varying vec4 diffuse,ambientGlobal,ambient;
varying vec3 normal,lightDir,halfVector;
varying float dist;
void main( )
{
    vec3 n,halfV,viewV,ldir;
    float NdotL,NdotHV;
    vec4 color = ambientGlobal;
    float att;
    n = normalize(normal);
    NdotL = max(dot(n,normalize(lightDir)),0.0);
    if (NdotL > 0.0)
    {
        att = 1.0 / (gl_LightSource[0]. constantAttenuation +
                gl_LightSource[0]. linearAttenuation * dist +
                gl_LightSource[0]. quadraticAttenuation * dist * dist);
        color += att * (diffuse * NdotL + ambient);
        halfV = normalize(halfVector);
        NdotHV = max(dot(n,halfV),0.0);
        color += att * gl_FrontMaterial. specular *
                        gl_LightSource[0]. specular *
                        pow(NdotHV,gl_FrontMaterial. shininess);
    }
    gl_FragColor = color;
}
```

3. 光源参数设置

GLSL 程序使用的光源参数是 OpenGL 的状态，因此需要在 OpenGL 程序里面进行相应的光源设置。例如，把第 7 个光源设置为点光源的代码如下。

```
GL. Enable(EnableCap. Light7);
GL. Light(LightName. Light7,LightParameter. Diffuse,
        GlobalSetting. Light7Setting. DiffuseColor);
GL. Light(LightName. Light7,LightParameter. Specular,
        GlobalSetting. Light7Setting. SpecularColor);
```

```
GL. Light( LightName. Light7 , LightParameter. Ambient ,
        GlobalSetting. Light7Setting. AmbientColor) ;

Vector4 pos =
new Vector4 ( (float) GlobalSetting. Light7Setting. LightPosition. x ,
        (float) GlobalSetting. Light7Setting. LightPosition. y ,
        (float) GlobalSetting. Light7Setting. LightPosition. z ,
        1. 0f) ;
GL. Light( LightName. Light7 , LightParameter. Position ,
        pos) ;
GL. Light( LightName. Light7 , LightParameter. LinearAttenuation ,
        GlobalSetting. Light7Setting. LinearAttenuation) ;
GL. Light( LightName. Light7 , LightParameter. ConstantAttenuation ,
        GlobalSetting. Light7Setting. ConstantAttenuation) ;
GL. Light( LightName. Light7 , LightParameter. QuadraticAttenuation ,
        GlobalSetting. Light7Setting. QuadraticAttenuation) ;
```

4. 效果图

如图4-9所示，点光源距离三维模型的距离由近到远。从中可以看出，虽然三维模型不变，但随着光源位置的远离，三维球面上受到的光照距离越来越大。

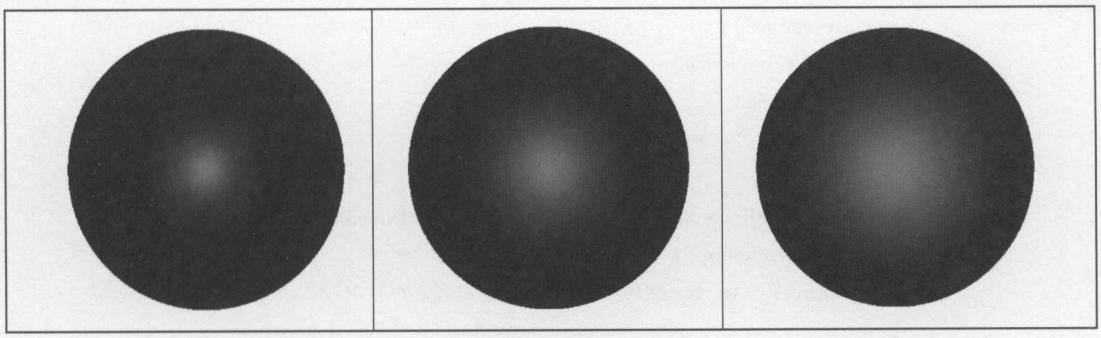


图4-9　点光源效果

4.6.2　聚光灯

聚光不同于点光源，其发出的光线被限制在一个圆锥体中。

（1）聚光包含一个方向向量 spotDirection，表示圆锥体的轴。

（2）圆锥体包含一个角度 spotCosCutoff。

（3）衰减速率 spotExponent 表示从圆锥的中心轴向外表面变化时光强度的衰减。

聚光光照计算与点光非常相似，但衰减必须乘以聚光效果（Spotlight Effect）

$$\text{spotEffect} = (\text{spotDirection} \cdot \text{lightDir})^{\text{spotExp}}$$

式中，spotDirection 是 OpenGL 中设置的状态；lightDir 是光源到某点的向量；spotExp 是聚光衰减率，这个值在 OpenGL 中设置，用来控制从聚光光锥中心到边缘的衰减，spotExp 越大衰

减越快，如果为0表示在光锥内光强是常数。

当光源与某点连线向量及聚光方向向量（spotDirection）之间夹角的余弦值小于 spotCosCutoff 时，这个顶点位于聚光灯的圆锥体之外，只能接收到全局环境光。只有当顶点位于聚光的光锥内时，才需要对散射光、镜面反射光和环境光分量进行着色。

聚光灯的代码和点光源类似，只需对像素着色器进行如下的条件判断。

```
spotEffect = dot( normalize( gl_LightSource[0]. spotDirection) ,
            normalize( -lightDir) ) ;
if ( spotEffect  >  gl_LightSource[0]. spotCosCutoff)
```

1. 顶点着色器

聚光灯顶点着色器代码和点光源一样，像素着色器代码如下。

```
varying vec4 diffuse,ambientGlobal,ambient;
varying vec3 normal,lightDir,halfVector;
varying float dist;
void main( )
{
    vec3 n,halfV,viewV,ldir;
    float NdotL,NdotHV;
    vec4 color = ambientGlobal;
    float att,spotEffect;
    n = normalize( normal) ;
    NdotL = max( dot( n,normalize( lightDir) ) ,0. 0) ;
    n = normalize( normal) ;
    if( NdotL  >  0. 0)
    {
        spotEffect = dot( normalize( gl_LightSource[0]. spotDirection) ,
                    normalize( -lightDir) ) ;
        if ( spotEffect  >  gl_LightSource[0]. spotCosCutoff)
        {
            att = 1. 0/( gl_LightSource[0]. constantAttenuation  +
                    gl_LightSource[0]. linearAttenuation * dist  +
                    gl_LightSource[0]. quadraticAttenuation
                    *  dist * dist) ;
            color  +  = att * ( diffuse * NdotL  +  ambient) ;
            halfV = normalize( halfVector) ;
            NdotHV = max( dot( n,halfV) ,0. 0) ;
            color  +  = att * gl_FrontMaterial. specular
            *  gl_LightSource[0]. specular  *
                        pow( NdotHV,gl_FrontMaterial. shininess) ;
        }
```

```
        }
        gl_FragColor = color;
    }
```

2. 光源参数设置

和方向光、点光源一样，必须在 OpenGL 设置相应的光照参数，才可以传递给 GLSL 进行使用。例如，下面把 OpenGL 的第 7 个灯光设置为聚光灯光源。那么在 GLSL 程序里面就需要相应地使用第 7 个灯光。如果不用 OpenGL 状态传递光照参数，就需要自定义 Uniform 变量来传递这些参数。

```
GL.Enable(EnableCap.Light6);
Vector4 pos =
    new Vector4((float)GlobalSetting.Light6Setting.LightPosition.x,
                (float)GlobalSetting.Light6Setting.LightPosition.y,
                (float)GlobalSetting.Light6Setting.LightPosition.z,
                1.0f);
GL.Light(LightName.Light6,LightParameter.Position,pos);
GL.Light(LightName.Light6,LightParameter.Diffuse,
        GlobalSetting.Light6Setting.DiffuseColor);
GL.Light(LightName.Light6,LightParameter.Specular,
        GlobalSetting.Light6Setting.SpecularColor);
GL.Light(LightName.Light6,LightParameter.Ambient,
        GlobalSetting.Light6Setting.AmbientColor);

GL.Light(LightName.Light6,LightParameter.LinearAttenuation,
        GlobalSetting.Light6Setting.LinearAttenuation);
GL.Light(LightName.Light6,LightParameter.ConstantAttenuation,
        GlobalSetting.Light6Setting.ConstantAttenuation);
GL.Light(LightName.Light6,LightParameter.QuadraticAttenuation,
        GlobalSetting.Light6Setting.QuadraticAttenuation);

Vector4 spotDirection =
    new Vector4((float)GlobalSetting.Light6Setting.SpotDirection.x,
                (float)GlobalSetting.Light6Setting.SpotDirection.y,
                (float)GlobalSetting.Light6Setting.SpotDirection.z,
                0.0f);
GL.Light(LightName.Light6,LightParameter.SpotDirection,
        spotDirection);
GL.Light(LightName.Light6,LightParameter.SpotExponent,
        GlobalSetting.Light6Setting.SpotExponent);
GL.Light(LightName.Light6,LightParameter.SpotCutoff,
        GlobalSetting.Light6Setting.SpotCutoff);
```

3. 效果图

如图 4-10 所示是各种三维模型在不同参数聚光灯照射下的效果图。图中圆形的部分就是聚光灯照射的范围。由于只使用了一个聚光灯光源，因此没有照射到的部分就呈现为黑色。通常聚光灯需要和其他类型的光源联合使用。

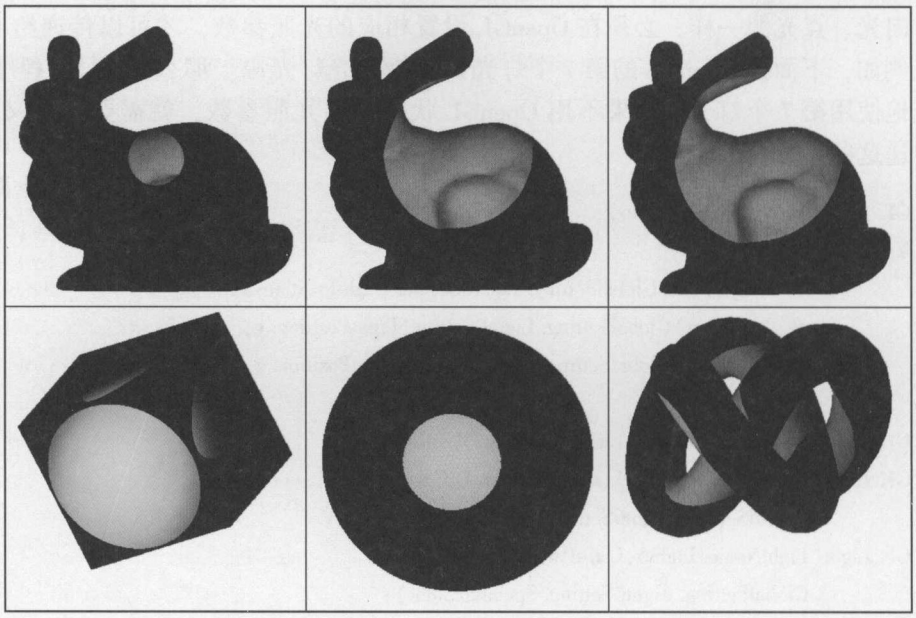


图 4-10　聚光灯光源效果

4.6.3　双面光照

三维模型有内外两个表面，渲染时可以选择只渲染一个表面，或者两个表面都进行渲染。双面光照中需要用到三维模型物体材质的参数，也就是需要给材质设置正面和背面的颜色等参数。然后 GLSL 可以利用这些参数来进行双面光照的计算。

1. 材质参数

材质参数是在 OpenGL 里面设置的，GLSL 可以通过内置的变量 gl_ FrontMaterial 直接得到相应的参数值。这样的参数传递只是为了方便使用，也可以自定义其他的 Uniform 变量来传递材质正反面参数。

```
public void SetMaterial( )
{
    GL. Material( MaterialFace. Front, MaterialParameter. Ambient,
                  GlobalSetting. MaterialSetting. MaterialAmbient);
    GL. Material( MaterialFace. Front, MaterialParameter. Emission,
                  GlobalSetting. MaterialSetting. MaterialEmission);
    GL. Material( MaterialFace. Front, MaterialParameter. Diffuse,
                  GlobalSetting. MaterialSetting. MaterialDiffuse);
```

```
        GL.Material(MaterialFace.Front,MaterialParameter.Specular,
                    GlobalSetting.MaterialSetting.MaterialSpecular);
        GL.Material(MaterialFace.Front,MaterialParameter.Shininess,
                    GlobalSetting.MaterialSetting.MaterialShininess);

        GL.Material(MaterialFace.Back,MaterialParameter.Ambient,
                    GlobalSetting.MaterialSetting.BackMaterialAmbient);
        GL.Material(MaterialFace.Back,MaterialParameter.Emission,
                    GlobalSetting.MaterialSetting.BackMaterialEmission);
        GL.Material(MaterialFace.Back,MaterialParameter.Diffuse,
                    GlobalSetting.MaterialSetting.BackMaterialDiffuse);
        GL.Material(MaterialFace.Back,MaterialParameter.Specular,
                    GlobalSetting.MaterialSetting.BackMaterialSpecular);
        GL.Material(MaterialFace.Back,MaterialParameter.Shininess,
                    GlobalSetting.MaterialSetting.BackMaterialShiness);
    }
```

材质参数界面如图4-11所示。

BackMaterialAmbient	Black
BackMaterialDiffuse	Tomato
BackMaterialEmission	Black
BackMaterialShiness	128
BackMaterialSpecular	Black
Enable	True
MaterialAmbient	Black
MaterialDiffuse	CornflowerBlue
MaterialEmission	Black
MaterialShininess	128
MaterialSpecular	Black

图4-11　材质参数界面

2. 单面光照

```
void OneSideLighting()
{
    vec3 n = normalize(normal);
    vec4 ambient  = vec4(0.0);
    vec4 diffuse  = vec4(0.0);
    vec4 specular = vec4(0.0);
    calculateLighting(gl_MaxLights,n,vertex,gl_FrontMaterial.shininess,
                      ambient,diffuse,specular);
```

```
    vec4 color = gl_FrontLightModelProduct. sceneColor    +
                 (ambient    * gl_FrontMaterial. ambient)  +
                 (diffuse    * gl_FrontMaterial. diffuse)  +
                 (specular * gl_FrontMaterial. specular);
    color = clamp(color,0. 0,1. 0);
    gl_FragColor = color;
}
```

3. 双面光照

```
void TwoSideLighting( )
{
    vec3 n = normalize(normal);
    vec4 ambient,diffuse,specular,color;
    ambient    = vec4(0. 0);
    diffuse    = vec4(0. 0);
    specular = vec4(0. 0);
    calculateLighting(gl_MaxLights,n,vertex,gl_FrontMaterial. shininess,
                      ambient,diffuse,specular);
    color    = gl_FrontLightModelProduct. sceneColor    +
               (ambient    * gl_FrontMaterial. ambient)  +
               (diffuse    * gl_FrontMaterial. diffuse)  +
               (specular * gl_FrontMaterial. specular);
    ambient = vec4(0. 0);
    diffuse = vec4(0. 0);
    specular = vec4(0. 0);
    calculateLighting(gl_MaxLights, -n,vertex,gl_BackMaterial. shininess,
                      ambient,diffuse,specular);
    color + = gl_BackLightModelProduct. sceneColor    +
              (ambient    * gl_BackMaterial. ambient)  +
              (diffuse    * gl_BackMaterial. diffuse)  +
              (specular * gl_BackMaterial. specular);
    color = clamp(color,0. 0,1. 0);
    gl_FragColor = color;
}
```

4.7 纹理贴图

三维模型渲染时除了设置光照、颜色之外，最重要的技术是纹理贴图技术。通过纹理贴图可以给同一个三维模型表面覆盖各种不同的图案。从而更加丰富三维模型的显示渲染效果。纹理贴图技术要求三维模型具有纹理坐标，也就是三维模型的每个顶点都具有一个二维坐标。纹理坐标把三维模型上的顶点映射到图片上，从而得到每个顶点相对应的颜色。三维

模型要显示出纹理贴图的效果，需要用 OpenGL 函数把纹理坐标传递给 GLSL 语言的内置变量 gl_ MultiTexCoord0。也就是除了输出顶点坐标和法向之外，还需要输出纹理坐标。纹理坐标内置变量 gl_ MultiTexCoord0 只能在顶点着色器中得到，因此需要把这个内置变量赋值给 gl_ TexCoord [0] 变量，从而可以把纹理坐标传递给像素着色器。三维模型最终显示的颜色由贴图的颜色和三维模型本身的颜色决定。贴图的颜色和顶点的光照颜色两种颜色有如表 4-1 所示几种混合方式。

表 4-1　几种混合方式

参　数	含　义
GL_REPLACE	贴图颜色替换三维模型颜色
GL_MODULATE	贴图颜色和三维模型颜色相乘
GL_DECAL	贴图颜色和三维模型颜色根据透明值进行混合
GL_BLEND	贴图环境颜色和三维模型颜色根据贴图颜色进行混合
GL_ADD	贴图颜色和三维模型颜色相加，透明度相乘
GL_COMBINE	贴图颜色和三维模型颜色相乘，然后归一化到 0 ~ 1 之间。

1. OpenGL 传递纹理坐标

这段代码把顶点坐标，顶点法向，顶点的纹理坐标从 OpenGL 传递到 GLSL。如果不用 GL. TexCoord2 函数来传递纹理坐标，那么 GLSL 里面的内置变量 gl_ MultiTexCoord0 就无法得到纹理坐标的值。从而不能用 GLSL 进行纹理贴图的显示。也就是 GLSL 里面的 Uniform 类型的内置变量都是通过 OpenGL 外部传递进来的。

```
public void DrawVertexUV(TriMesh mesh)
{
    GL. Begin(BeginMode. Triangles);
    for (int i=0; i < mesh. Faces. Count; i++)
    {
        foreach (TriMesh. Vertex vertex in mesh. Faces[i]. Vertices)
        {
            GL. Normal3(vertex. Traits. Normal. x,
                        vertex. Traits. Normal. y,
                        vertex. Traits. Normal. z);
            GL. TexCoord2(vertex. Traits. UV[0],vertex. Traits. UV[1]);
            GL. Vertex3(vertex. Traits. Position. x,
                        vertex. Traits. Position. y,
                        vertex. Traits. Position. z);
        }
    }
    GL. End();
}
```

2. 顶点着色器

顶点着色器完成得到纹理坐标，并把纹理坐标传递给像素着色器。

```
void main( )
{
    gl_TexCoord[0] = gl_MultiTexCoord0;
    gl_Position = ftransform( );
}
```

3. 像素着色器

像素着色器根据纹理坐标和三维模型当前的颜色来得到最终的像素颜色。其中 Uniform 变量 sampler2D 是从外部传递进来的贴图。假如不用 GLSL 进行渲染的话，那么 OpenGL 内置函数会对这些纹理坐标进行处理。但是用 GLSL 进行渲染，就需要编写相应的 GLSL 程序来处理纹理坐标和贴图之间的映射来得到每个像素的颜色值。

```
uniform sampler2D TexUnit0;
uniform int TexturingType;
void main( )
{
    vec4 color = gl_Color;
    applyTexture2D( TexUnit0 , TexturingType0 , 0 , color) ;
    gl_FragColor = color;
}
```

4. 混合方式函数

这个函数计算 6 种贴图颜色和三维模型本身颜色混合的结果。这 6 种方式是 OpenGL 标准定义的。但因为用 GLSL 进行渲染，那么还可以实现更多的自定义的混合方式。

```
const int REPLACE   =0;
const int MODULATE =1;
const int DECAL     =2;
const int BLEND     =3;
const int ADD       =4;
const int COMBINE   =5;

void applyTexture2D( in sampler2D texUnit , in int type ,
                     in int index , inout vec4 color)
{
    vec4 texture = texture2D( texUnit , gl_TexCoord[ index ]. st) ;
    if ( type == REPLACE)
    {
        color = texture;
    }
    else if ( type == MODULATE)
    {
        color  * = texture;
```

```
    }
    else if (type == DECAL)
    {
        vec3 temp = mix(color.rgb, texture.rgb, texture.a);

        color = vec4(temp, color.a);
    }
    else if (type == BLEND)
    {
        vec3 temp = mix(color.rgb,
                gl_TextureEnvColor[index].rgb, texture.rgb);

        color = vec4(temp, color.a * texture.a);
    }
    else if (type == ADD)
    {
        color.rgb  += texture.rgb;
        color.a    *= texture.a;

        color = clamp(color, 0.0, 1.0);
    }
    else
    {
        color = clamp(texture * color, 0.0, 1.0);
    }
}
```

5. 多重纹理

多重纹理是指同时使用两个或者两个以上的纹理贴图，相应的代码如下。

```
void MultiTexture()
{
    vec4 color = gl_Color;
    applyTexture2D(TexUnit0, TexturingType0, 0, color);
    applyTexture2D(TexUnit1, TexturingType1, 0, color);
    gl_FragColor = color;
}
```

6. 纹理贴图传递

在GLSL程序编写好之后，需要从外部连接上相应的GLSL程序，以及把纹理贴图的ID号传递到GLSL程序。因此在OpenGL加载纹理贴图并生成ID之后，进行贴图传递的具体程序如下。

```
public class GPUSingleTexture : AbstractGPURender
{
    public GPUSingleTexture()
    {
        VSFileName = "GLSL/Texturing.vert";
        FSFileName = "GLSL/SingleTexture.frag";
    }
    public override void PassParameter()
    {
        GL.Uniform1(GL.GetUniformLocation(this.handle,"TexUnit0"),
            OpenGLManager.Instance.TextureList[0].ID);
        GL.Uniform1(GL.GetUniformLocation(this.handle,"TexUnit1"),
            OpenGLManager.Instance.TextureList[1].ID);
        GL.Uniform1(GL.GetUniformLocation(this.handle,"TexturingType0"),
            ConfigGPUSingleTexture.Instance.TypeHybrid);
        GL.Uniform1(GL.GetUniformLocation(this.handle,"TexturingType1"),
            ConfigGPUSingleTexture.Instance.TypeHybrid);
        GL.Uniform1(GL.GetUniformLocation(this.handle,"Single"),
            ConfigGPUSingleTexture.Instance.Single);

    }
}
```

7. 效果图

1）实验一

如图 4-12 所示是各种不同三维模型用 GLSL 进行贴图的渲染效果。

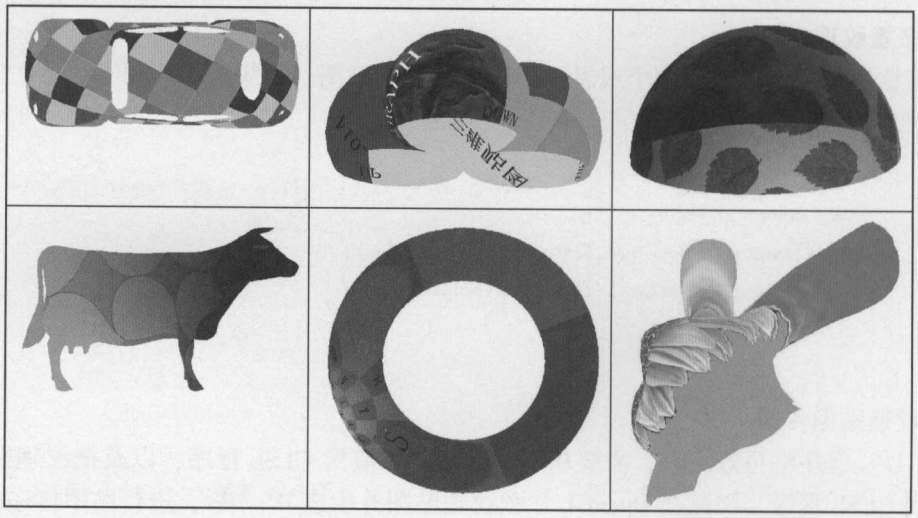


图 4-12　渲染效果

2）实验二

如图 4-13 所示是在多种不同混合模式下，贴图和三维模型本身颜色的混合效果。可以看出根据混合方法的不同，最终的显示结果不一样。最重要的是可以根据需求来编写 GLSL 程序得到特定的混合结果，而不是受限于 OpenGL 本身提供的混合方式。

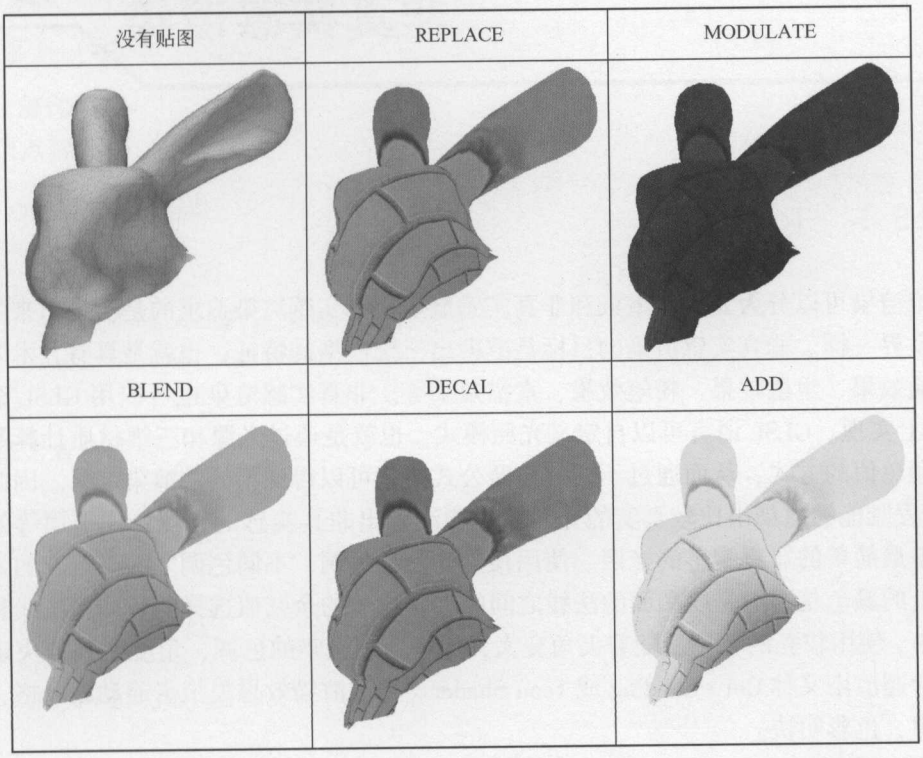


图 4-13　贴图混合方式

第5章 非真实感渲染

5.1 卡通渲染

三维渲染可以分为真实感渲染和非真实感渲染。真实感渲染追求的是渲染出来的图像和真实的世界一样。非真实感渲染的目标是渲染出三维模型的特征，也就是具有艺术效果。例如，卡通效果、水墨效果、粉笔效果、素描效果等。非真实感渲染也可以用 GLSL 语言编程在 GPU 上实现。GLSL 语言可以自定义光照模式，也就是通过光源和三维材质计算最终每个像素的颜色值的方式，从而通过不同的光照公式，就可以得到不同的渲染结果。因此，通过 GLSL 编程既能够渲染出比较真实的光照，也能渲染出非真实感的光照，如卡通等效果。卡通着色是最简单的非真实感的光照。使用很少的几种色调，不同色调之间是突变的效果。三维模型上的颜色是通过光线和面的法线之间的夹角角度的余弦值选择的。如果法线和光的夹角比较小，使用较亮的色调，随着夹角变大，逐步使用更暗的色调，角度余弦值决定色调的强度。卡通渲染又称 Cel - Shading 或 Toon Shader。卡通渲染效果模拟卡通动画风格，带有黑色的轮廓，色彩明快。

1. 算法步骤

1）顶点着色器

第一步：计算法向量。因为涉及光照计算，所以在顶点着色器中要计算顶点的法向量。

```
Normal = normalize( gl_NormalMatrix * gl_Normal) ;
```

第二步：计算顶点位置。

```
gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
```

2）像素着色器

像素着色器要根据光线和面的法线之间夹角角度的余弦值的大小实现卡通效果的黑色轮廓和光照渲染。

第一步：计算光线和面的法线之间的夹角角度的余弦值。

```
float intensity = dot( LightPosition , Normal) ;
```

第二步：判断余弦值如果小于给定的阈值，则渲染为黑色轮廓。

```
if ( abs( intensity) < Edge)
    color = vec3(0) ;
```

第三步：判断余弦值如果大于给定的阈值，则渲染为给定的颜色值。

```
if (intensity > Phong)
    color = PhongColor;
```

第四步：计算最终着色颜色值。

```
gl_FragColor = vec4(color,1);
```

2. 完整代码

1）顶点着色器

```
varying vec3 Normal;
void main(void)
{
    Normal = normalize(gl_NormalMatrix * gl_Normal);
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

2）像素着色器

```
uniform vec3 DiffuseColor;
uniform vec3 PhongColor;
uniform float Edge;
uniform float Phong;
varying vec3 Normal;
uniform vec3 LightPosition;

void main (void)
{
    vec3 color = DiffuseColor;
    float intensity = dot(LightPosition,Normal);
    if (abs(intensity) < Edge)
        color = vec3(0);
    if (intensity > Phong)
        color = PhongColor;
    gl_FragColor = vec4(color,1);
}
```

3. 参数界面

卡通效果 GLSL 程序里面用到的参数由于不是 OpenGL 内置的变量，因此需要进行自定义，然后通过菜单界面进行修改。界面如图 5-1 所示。

Misc	
DiffuseColor	0, 63, 255
Edge	0.5
Phong	0.98
PhongColor	191, 191, 255

图 5-1　卡通渲染参数界面

4. 参数传递程序

卡通渲染效果需要从 C#语言向 GLSL 语言传递参数。定义的卡通渲染效果类代码如下。其分为两部分：第一部分指定卡通渲染效果的顶点着

色器和像素着色器代码文件位置；第二部分把 GLSL 代码需要的参数传递进去。

```
public class GPURenderToon:AbstractGPURender
{
    public GPURenderToon()
    {
        VSFileName = "GLSL/NPR/Toon.vert";
        FSFileName = "GLSL/NPR/Toon.frag";
    }
    public override void PassParameter()
    {
        GL.Uniform3(GL.GetUniformLocation(this.handle,"LightPosition"),
            ConfigGPUCommon.Instance.LightPosition);
        GL.Uniform3(GL.GetUniformLocation(this.handle,"DiffuseColor"),
            ConfigGPUToon.Instance.DiffuseColor );
        GL.Uniform3(GL.GetUniformLocation(this.handle,"PhongColor"),
            ConfigGPUToon.Instance.PhongColor );
        GL.Uniform1(GL.GetUniformLocation(this.handle,"Edge"),
            ConfigGPUToon.Instance.Edge);
        GL.Uniform1(GL.GetUniformLocation(this.handle,"Phong"),
            ConfigGPUToon.Instance.Phong);
    }
}
```

5. 效果图

如图 5-2 所示是三维模型在不同参数下的卡通效果。可以看出三维模型的线条被凸显出来。通过线条和颜色的对比从而得到卡通效果显示。

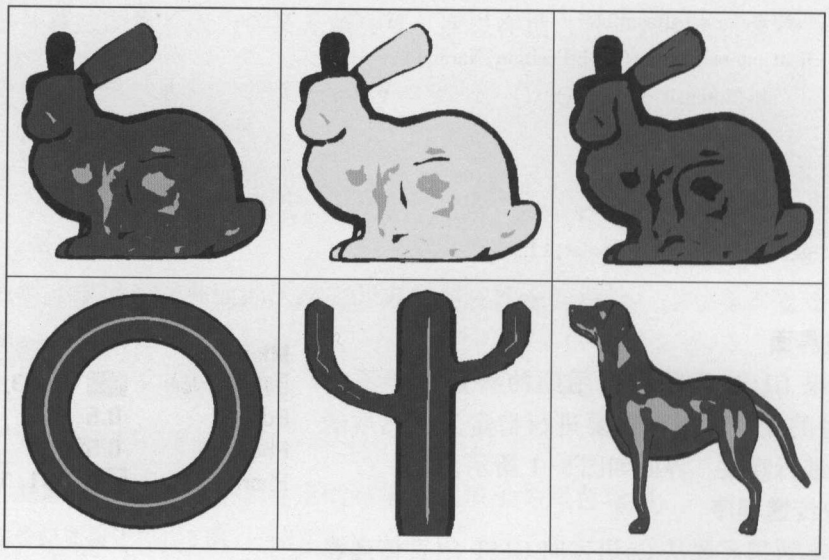


图 5-2　卡通渲染效果

5.2　影线渲染

影线（Hatch）渲染是一种渲染出来像手绘的非真实感渲染效果。影线渲染的目标是得到和用钢笔或墨水线条绘制出木刻版画类似的效果。影线渲染效果需要向着色器传递顶点位置、法向量及纹理坐标。顶点位置是每一个着色器都必须具有的，而法向量是计算光照的基础，纹理坐标是用来制作影线的基础。可以引入噪声函数来丰富渲染效果，加入噪声的方式是将噪声值存储在一个3D纹理中。建立影线的思路首先将纹理坐标的 t 分量乘以一个常量并取分数部分，可以创建一种锯齿形波浪。根据内置函数的图形可以知道这将导致函数值从0开始，增长到1，然后突然回到0，这个序列将会重复frequency次，得到的效果图如图5-3（a）所示。实现代码如下。

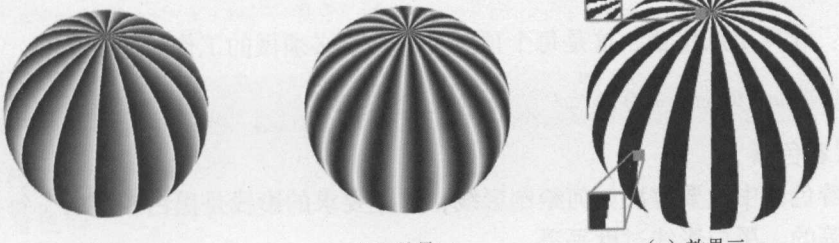
```
float sawtooth = fract( V frequency) ;
```

但这并不是期望的结果，接下来需要用三角波来加强效果。绝对值函数得到一个从1减到0，然后又增加到1的函数，也就是一个三角波，效果如图5-3（b）所示。相应的代码如下。

```
float triangle = abs( 2.0 * sawtooth - 1.0) ;
```

进行到这一步虽然可以得到影线的效果，但是比较模糊。接下来用step函数，将影线变成纯黑和纯白。用三角波变量和0.5进行比较。如图5-3（c）所示。代码如下。

```
Float sqaure = step( 0.5 , triangle) ;
```



（a）效果一　　（b）效果二　　（c）效果三

图5-3　影线生成图

现在能比较清晰地看到明暗相间的影线了。但问题是影线是不等宽的，极点处窄、中心宽，而且step函数的返回值只有0或1，没有中间值，因此黑白之间的过渡区域会出现锯齿，所以要实现影线的等宽并消除锯齿现象，还要增加某些特征来达到更加真实的影线效果。

1. 算法步骤

1）顶点着色器

第一步：变量说明。

```
uniform vec3    LightPosition;
uniform float Time;
```

```
varying vec3  ObjPos;
varying float V;
varying float LightIntensity;
```

第二步：计算噪声纹理的坐标。为了实现影线扭动的动画效果，向顶点的 Z 坐标添加了一致变量 Time，这样会看起来像沿着 Z 轴扭动。

```
ObjPos   = (vec3(gl_Vertex) + vec3(0.0,0.0,Time)) * 0.2;
```

第三步：计算光线强度。在前面已经学习了相关知识，根据入射光线和法线可以计算漫反射强度。

```
vec3 pos       = vec3(gl_ModelViewMatrix * gl_Vertex);
vec3 tnorm     = normalize(gl_NormalMatrix * gl_Normal);
vec3 lightVec = normalize(LightPosition - pos);
LightIntensity = max(dot(lightVec,tnorm),0.0);
```

第四步：确定绘制影线的方法。这里使用一个单独的坐标参数作为影线的基础，并使用易变变量 V 传入，假定是对象纹理坐标的 t 坐标。假如没有纹理坐标，那么就使用顶点坐标的值。

```
if(gl_MultiTexCoord0.s==0&&gl_MultiTexCoord0.t==0)
    V = gl_Vertex.y;
else
    V = gl_MultiTexCoord0.t;
```

第五步：计算定点位置，这是每个顶点着色器都必须做的工作。

```
gl_Position = ftransform();
```

2）像素着色器

在像素着色器中主要解决如何绘制影线，这里要求的影线是黑白相间的、每条影线的宽度是大致相等的、黑白影线过度平滑。

第一步：变量说明。

```
const float frequency = 1.0;

varying vec3  ObjPos;
varying float V;
varying float LightIntensity;

uniform sampler3D Noise;
uniform float Swidth;
```

第二步：绘制等宽度的影线。梯度函数能够计算在某一点上的变化率，可以用梯度大小来确定影线密度。除此之外，必须在密度太大时减小影线数量，这可以通过取对数的负数得到。

```
float dp = length( vec2( dFdx( V * Swidth) ,dFdy( V * Swidth) ) ) ;
float logdp     =  - log2( dp) ;
float ilogdp    = floor( logdp) ;
float stripes   = exp2( ilogdp) ;
float noise     = texture3D( Noise, ObjPos) . x;
float sawtooth  = fract( V * frequency * stripes) ;
```

第三步：平滑影线间过渡时的生硬变化。使用变量 transition 在三角波的两个频率之间做一次平滑，使用线性插值的方法来柔化边缘。

```
float triangle = abs( 2. 0 * sawtooth - 1. 0) ;
float transition = logdp - ilogdp;
triangle = abs( ( 1. 0 + transition) * triangle - transition) ;
```

第四步：添加光照。使用光照强度修改函数中的阈值，可以实现阴影区域的暗影线更突出一些，明亮区域的白影线更突出一些。在明亮的区域需要减小阈值使黑色条纹更窄，在阴影区域要增加阈值使黑色条纹更宽。这里使用 smoothstep 函数消除过渡处的走样。

```
const float edgew = 0. 2;
float edge0    = clamp( LightIntensity - edgew, 0. 0, 1. 0) ;
float edge1    = clamp( LightIntensity, 0. 0, 1. 0) ;
float square = 1. 0 - smoothstep( edge0, edge1, triangle) ;
```

第五步：添加噪声特征。为了使影线效果更加真实，所以需要给影线添加一点扭曲效果或者是添加噪点，为此要修改生成影线的函数。

```
float sawtooth = fract( ( V + noise * 0. 1) * frequency * stripes) ;
```

第六步：计算片元值。

```
gl_FragColor = vec4( vec3( square) ,1. 0) ;
```

2. 完整代码

1）顶点着色器

```
uniform vec3   LightPosition;
uniform float Time;
varying vec3   ObjPos;
varying float V;
varying float LightIntensity;

void main( )
{
    ObjPos          = ( vec3( gl_Vertex) + vec3( 0. 0, 0. 0, Time) ) * 0. 2;
    vec3 pos        = vec3( gl_ModelViewMatrix * gl_Vertex) ;
    vec3 tnorm      = normalize( gl_NormalMatrix * gl_Normal) ;
    vec3 lightVec   = normalize( LightPosition - pos) ;
```

```
    LightIntensity    = max( dot( lightVec, tnorm) ,0.0) ;
    if( gl_MultiTexCoord0. s ==0&&gl_MultiTexCoord0. t ==0)
        V = gl_Vertex. y;
    else
        V = gl_MultiTexCoord0. t;
    gl_Position = ftransform( ) ;
}
```

2）像素着色器

```
const float frequency = 1.0;
varying vec3   ObjPos;
varying float V;
varying float LightIntensity;
uniform sampler3D Noise;
uniform float Swidth;

void main( )
{
    float dp          = length( vec2( dFdx( V * Swidth) , dFdy( V * Swidth) ) ) ;
    float logdp       = - log2( dp) ;
    float ilogdp      = floor( logdp) ;
    float stripes     = exp2( ilogdp) ;
    float noise       = texture3D( Noise, ObjPos). x;
    float sawtooth    = fract( ( V + noise * 0.1) * frequency * stripes) ;
    float triangle = abs( 2.0 * sawtooth - 1.0) ;
    float transition = logdp - ilogdp;
    triangle = abs( ( 1.0 + transition) * triangle - transition) ;
    const float edgew = 0.2;
    float edge0    = clamp( LightIntensity - edgew, 0.0, 1.0) ;
    float edge1    = clamp( LightIntensity, 0.0, 1.0) ;
    float square = 1.0 - smoothstep( edge0, edge1, triangle) ;
    gl_FragColor = vec4( vec3( square) , 1.0) ;
}
```

3）参数传递

```
public class GPURenderHatch: AbstractGPURender
{
    public GPURenderHatch( )
    {
        VSFileName = "GLSL/NPR/hatch. vert" ;
        FSFileName = "GLSL/NPR/hatch. frag" ;
        PerlinNoise noise = new PerlinNoise( ) ;
```

```
            noise. CreateNoise3D( ) ;
        }
        public override void PassParameter( )
        {
            GL. Uniform3( GL. GetUniformLocation( this. handle, "LightPosition" ) ,
                ConfigGPUCommon. Instance. LightPosition) ;
            GL. Uniform1( GL. GetUniformLocation( this. handle, "Swidth" ) ,
                ConfigGPUHatch. Instance. Swidth) ;
            GL. Uniform1( GL. GetUniformLocation( this. handle, "Time" ) ,
                ConfigGPUHatch. Instance. Time) ;
            GL. Uniform1( GL. GetUniformLocation( this. handle, "Noise" ) ,  0) ;
        }
    }
```

3. 效果图

实验一：如图5-4所示是没有采用噪声纹理坐标得到影线渲染结果。第一行是不同模型的影线效果，第二行是同一个模型不同参数的阴线效果。

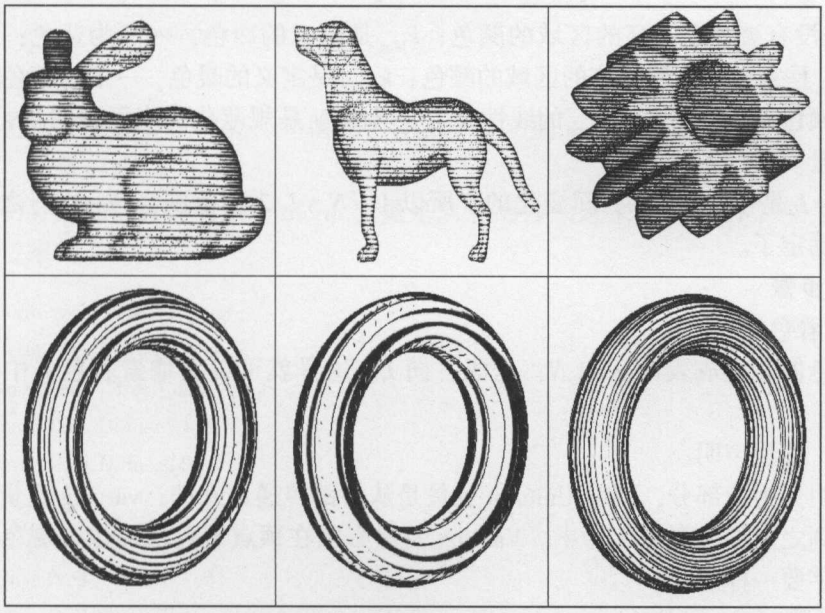


图5-4　影线渲染效果

实验二：采用噪声纹理得到的影线渲染效果，如图5-5所示。

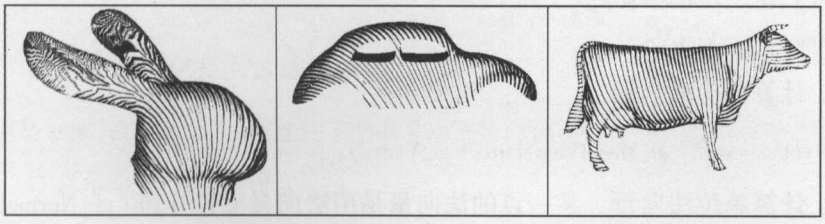


图5-5　噪声纹理影线渲染效果

5.3 Gooch 渲染

Gooch 渲染也是一种非真实感渲染，也是自定义了一种不同于 OpenGL 内置的光照模式。Gooch 渲染意在用简洁、清晰的方式传达信息，有些无益于观察者理解的细节会被忽略，而对三维模型形状理解很关键的细节会显示得非常清晰、直接。Gooch 渲染主要有三大特点：一是它会生成代表重要边缘的黑色曲线；二是使用镜面高光，并用白色进行渲染；三是亮度值的范围是有限的，用来传达所渲染表面的曲率信息，这里用暖色到冷色的梯度来表达亮度值的范围。三维模型的实际着色取决于两个方面：一方面是漫反射用来生成亮度值的范围提供一部分着色；另一方面是冷色和暖色之间过渡提供另一部分着色。

1. 渲染公式

用来计算 Gooch 渲染颜色公式如下。

$$k_{cool} = k_{blue} + \alpha k_{diffuse}$$

$$k_{warm} = k_{yellow} + \beta k_{diffuse}$$

$$k_{final} = \left[\frac{1 + N * L}{2}\right] k_{cool} + \left[1 - \frac{1 + N * L}{2}\right] k_{warm}$$

式中，k_{cool}是没有被光照照亮的区域的颜色；k_{blue}是定义的冷色，一般为蓝色；$k_{diffuse}$是漫反射颜色；k_{warm}是完全被光照着凉的区域的颜色；k_{yellow}是定义的暖色，一般为黄色。

最终的颜色 k_{final}是 k_{cool}和 k_{warm}的线性组合，其中 N 是规范化的表面法向量，L 是光源方向的单位矢量。

因为 $N * L$ 是在[-1,1]之间变化的，所以 $1 + N * L/2$ 的范围会在[0,1]之间，最终的着色就可以确定了。

2. 算法步骤

1）顶点着色器

顶点着色器要生成表面法向 N，光源方向 L。这样就可以在像素着色器中计算镜面反射了。

第一步：变量声明。

变量声明分为两部分，其中 Uniform 变量是从外部传递进来的，varying 变量是从顶点着色器中传递到之后的像素着色器中。Varying 变量必须在顶点着色器和像素着色器中同时声明，并且名字要一样。

```
uniform vec3  LightPosition;
varying float NdotL;
varying vec3  ReflectVec;
varying vec3  ViewVec;
```

第二步：计算变换后坐标。

```
vec3 ecPos = vec3 (gl_ModelViewMatrix * gl_Vertex);
```

第三步：计算单位法向量。某一点的法向量是用法向量变换矩阵(gl_NormalMatrix)变换内置法向量变量(gl_Normal)计算的，并用内置函数 normalize 规范化。

```
vec3 tnorm = normalize( gl_NormalMatrix * gl_Normal );
```

第四步：计算从渲染对象表面的当前点到光源位置的矢量。第一步中已经得到了视觉坐标，所以从光源位置减去对象的当前点位置就得到当前点到光源位置的矢量。

```
vec3 lightVec = normalize( LightPosition - ecPos );
```

第五步：计算反射矢量。调用内置函数 reflect，reflect 需要入射矢量，所以第四步得到的光线矢量要取反。

```
ReflectVec = normalize( reflect( -lightVec, tnorm ) );
```

第六步：计算查看位置方向上的矢量。

```
ViewVec    = normalize( -ecPos );
```

第七步：计算 $1 + N * L/2$ 因子。

```
NdotL    = ( dot( lightVec, tnorm ) + 1.0 ) * 0.5;
```

第八步：计算顶点位置。

```
gl_Position    = ftransform( );
```

2）像素着色器

像素着色器要实现着色功能，并增加了镜面反射功能。

第一步：变量声明。

变量声明分为两部分，其中 Uniform 变量是从外部传递进来的，varying 变量是从顶点着色器中传递过来的。

```
uniform vec3   SurfaceColor;
uniform vec3   WarmColor;
uniform vec3   CoolColor;
uniform float DiffuseWarm;
uniform float DiffuseCool;

varying float NdotL;
varying vec3   ReflectVec;
varying vec3   ViewVec;
```

第二步：计算没有被光照照亮的区域的颜色。

```
vec3 kcool      = min( CoolColor + DiffuseCool * SurfaceColor, 1.0 );
```

第三步：计算被光照照亮的区域的颜色。

```
vec3 kwarm      = min( WarmColor + DiffuseWarm * SurfaceColor, 1.0 );
```

第四步：计算最终的 k_{final} 颜色。

```
vec3 kfinal    = mix( kcool, kwarm, NdotL );
```

第五步：规范化插值后的反射矢量和查看位置矢量。

```
vec3 nreflect = normalize( ReflectVec) ;
vec3 nview    = normalize( ViewVec) ;
```

第六步：计算镜面反射。

```
float spec    = max( dot( nreflect, nview) ,0.0) ;
spec          = pow( spec,32.0) ;
```

第七步：计算 k_{final} 和镜面反射的叠加值。

```
gl_FragColor = vec4 ( min( kfinal + spec,1.0) ,1.0) ;
```

3. 完整代码

1）顶点着色器

```
uniform vec3  LightPosition;
varying float NdotL;
varying vec3  ReflectVec;
varying vec3  ViewVec;

void main( void)
{
    vec3 ecPos      =  vec3 ( gl_ModelViewMatrix * gl_Vertex) ;
    vec3 tnorm      = normalize( gl_NormalMatrix * gl_Normal) ;
    vec3 lightVec   = normalize( LightPosition - ecPos) ;
    ReflectVec      = normalize( reflect( - lightVec, tnorm)) ;
    ViewVec         = normalize( - ecPos) ;
    NdotL           = ( dot( lightVec, tnorm) + 1.0) * 0.5;
    gl_Position     = ftransform( ) ;
}
```

2）像素着色器

```
uniform vec3  SurfaceColor;
uniform vec3  WarmColor;
uniform vec3  CoolColor;
uniform float DiffuseWarm;
uniform float DiffuseCool;

varying float NdotL;
varying vec3  ReflectVec;
varying vec3  ViewVec;

void main ( void)
```

```
{
    vec3 kcool      =  min( CoolColor + DiffuseCool * SurfaceColor,1.0);
    vec3 kwarm      = min( WarmColor + DiffuseWarm * SurfaceColor,1.0);
    vec3 kfinal     = mix( kcool,kwarm,NdotL);

    vec3 nreflect   = normalize( ReflectVec);
    vec3 nview      = normalize( ViewVec);

    float spec      = max( dot( nreflect,nview),0.0);
    spec            = pow( spec,32.0);

    gl_FragColor = vec4 ( min( kfinal + spec,1.0),1.0);
}
```

4. 效果图

如图5-6所示是不同三维模型在不同参数下的Gooch渲染效果，从中可以看出渲染的结果展示出和OpenGL内置的光照模式不一样的效果。可以看出Gooch效果显示出冷暖色调的感觉。

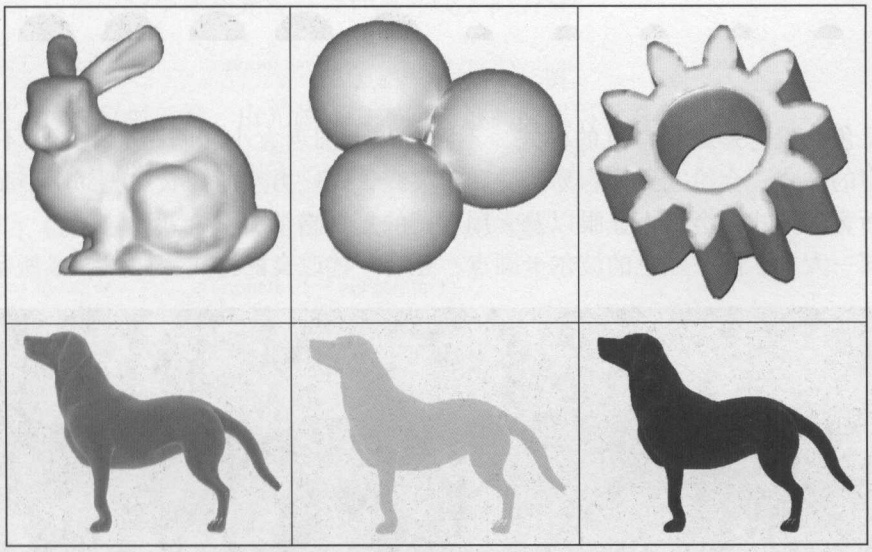


图5-6　Gooch渲染效果图

5.4　波尔卡圆点渲染

波尔卡圆点（Polka Dot）一般是同一大小、同一种颜色的圆点以一定的距离均匀地排列而成，其名字来源于一种名叫波尔卡的东欧音乐。在20世纪50年代，女人们身着蓬松的过膝裙装，以黑底白点作为配色的波尔卡圆点是最为时髦的图案。它带出一种轻松、轻盈、自由的气息，而这刚好是波尔卡圆点的精髓精神所在。常见的波尔卡圆点图案如图5-7所示。

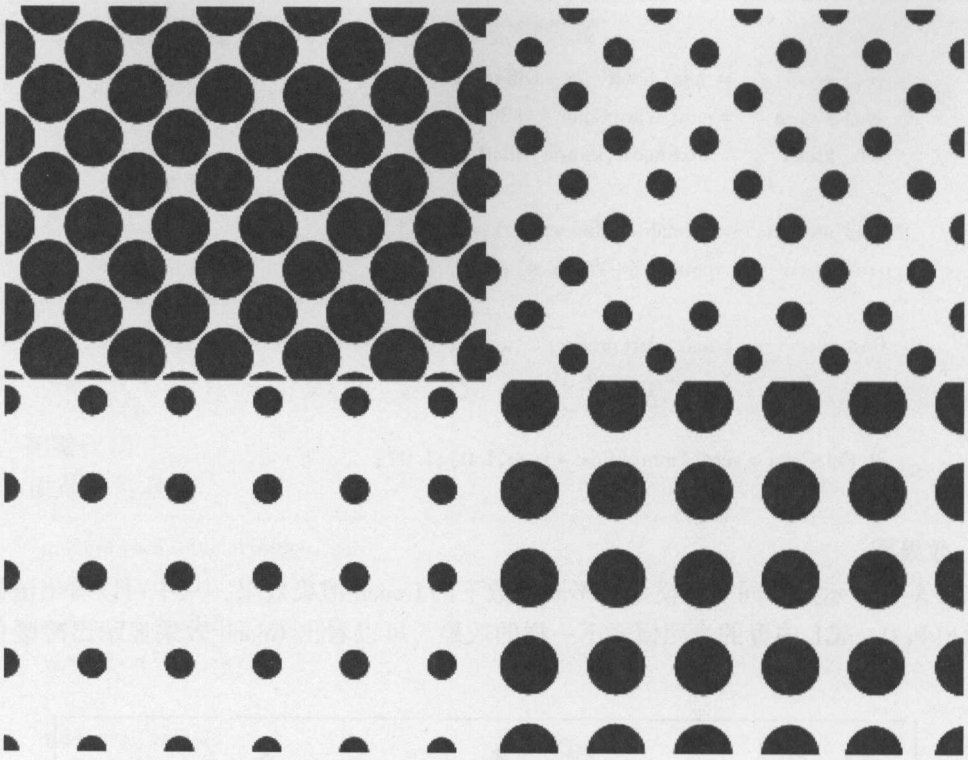


图 5-7　波尔卡圆点图案

黑白反色的设定是最为常见的样式，不同波尔卡圆点大小和间距会呈现很不一样的感觉。小而密的样式，会给人更为容易接受和不突兀，好比男生冲锋衣一般都用小而密的这种设计。大而宽的样式，会给人显眼以及无用质疑的复古感觉。

除了同一大小、同一颜色的波尔卡圆点，还有一些改良的样式，如图 5-8 所示。

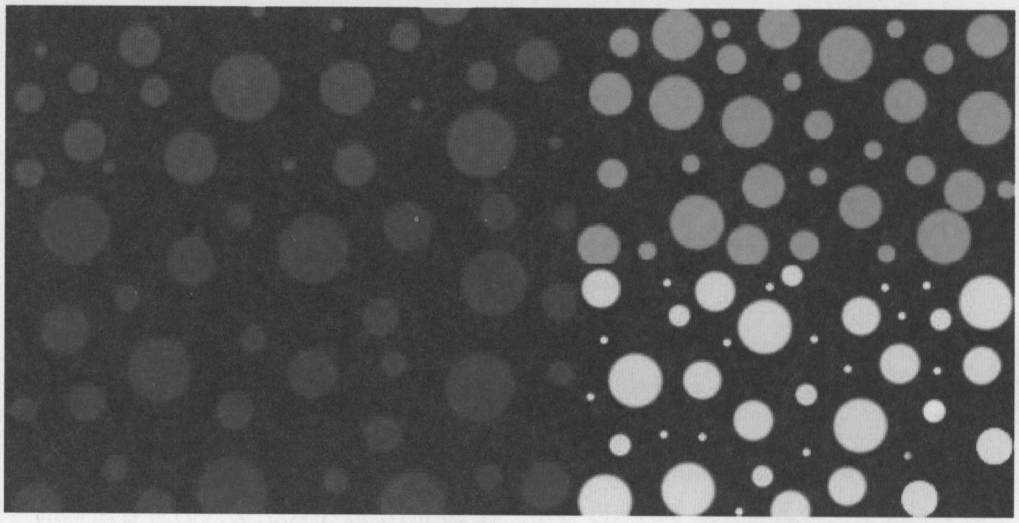


图 5-8　改良的波尔卡圆点图案

这类好比是把大小不同的波尔卡圆点混合在一起，以小区域作为重复单位，而不是单独的以一个波尔卡圆点作为重复单位。

PolkaDot3D 渲染效果是指在三维模型的表面生成波尔卡圆点图案，这与以往在二维图像上生成波尔卡圆点图案相比将会更加绚丽，更加富有应用价值，任何的 3D 模型都可以添加波尔卡圆点图案的设计。

GLSL 实现这个着色器的关键是如何在三维模型上绘制波尔卡圆点图案，这里选用采用阈值法，如果定义的波尔卡圆点的大小小于阈值则绘制图案，否则不绘制。绘制波尔卡圆点图案填充一种颜色，不绘制图案的填充另一种颜色，这样就能清晰地看到 PolkaDot3D 渲染效果。为了使渲染效果更加真实，这里使用了光照，顶点着色器主要负责光照计算，像素着色器负责波尔卡圆点图案绘制。

1. 算法步骤

1）顶点着色器

第一步：计算眼睛坐标和法向量。因为要进行光照计算，所以在顶点着色器中要计算顶点的法向量和眼睛坐标。

```
vec3 ecPosition    = vec3(gl_ModelViewMatrix * gl_Vertex);
vec3   tnorm       = normalize(gl_NormalMatrix * gl_Normal);
```

第二步：根据眼睛坐标和光源位置计算光线方向矢量。

```
vec3   lightVec    = normalize(LightPosition - ecPosition);
```

第三步：计算反射矢量。由光源入射方向矢量和法线可计算反射矢量

```
vec3 reflectVec    = reflect( -lightVec,tnorm);
```

第四步：根据眼睛坐标计算查看方向上的单位矢量。

```
vec3 viewVec       = normalize( -ecPosition);
```

第五步：根据计算漫反射强度并初始化镜面反射。

```
float  diffuse     = max(dot(lightVec,tnorm),0.0);
float  spec        =0.0;
```

第六步：根据计算镜面反射强度。

```
if(diffuse >0.0)
{
        spec = max(dot(reflectVec,viewVec),0.0);
        spec = pow(spec,16.0);
    }
```

第七步：计算最终的光照强度。

```
LightIntensity   = diffusecontribution * diffuse * 1.5  + SpecularContribution * spec;
```

第八步：计算顶点位置。在像素着色器绘制波尔卡圆点图案就是依据顶点位置设置阈值来决定是否在此处绘制图案的。

```
MCPosition = vec3 (gl_Vertex);
```

第九步：这是每一个顶点着色器都必须做的事情，计算齐次顶点位置。

```
gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
```

2）像素着色器

用户可以自行设置波尔卡圆点图案的大小和间隔及填充的颜色，所以这里设置了 uniform 变量，以下是变量说明。

```
varying float LightIntensity;
varying vec3 MCPosition;
uniform vec3 Spacing;
uniform float DotSize;
uniform vec3 ModelColor,PolkaDotColor;
```

第一步：计算是否绘制波尔卡圆点图案的阈值。这里的 scaledpoint 设置为三维向量，它的计算是根据在顶点着色器计算的顶点位置，每一个顶点都会有不同的判断标准。

```
Scaledpoint   =   MCPosition - (Spacing * floor(MCPosition/Spacing));
scaledpoint   =   scaledpoint - Spacing/2.0;
```

第二步：计算阈值大小。这里使用 OpenGL 内置函数 Length 来计算向量长度。

```
scaledpointlength   =   length(scaledpoint);
```

第三步：将波尔卡圆点图案大小和阈值进行比较。这里使用 step 函数，返回 0，1 表示两者之间的关系。

```
insidesphere   =   step(scaledpointlength,DotSize);
```

第四步：根据返回的 0，1 值确定填充的颜色，即绘制波尔卡圆点图案填充一种颜色，不绘制时填充另一种颜色。

```
finalcolor   =   vec3(mix(ModelColor,PolkaDotColor,insidesphere));
```

第五步：综合光照强度计算最终的着色值。

```
gl_FragColor   =   clamp(vec4(finalcolor * LightIntensity,1.0),0.0,1.0);
```

2. 参数界面

如图 5-9 所示是 PolkDot3D 用到的参数界面。其中参数 Dotsize 表示波尔卡圆点的大小，ModelColor 表示模型的颜色，PolkaDotColor 表示波尔卡圆点的颜色，Spacing 表示波尔卡圆点的间隔，通过调整参数可以得到不同的效果。

DotSize	0.1
ModelColor	(0.7490196, 0.2, 0.09803922)
PolkaDotColor	(1, 1, 1)
Spacing	(0.3, 0.3, 0.25)
SpecularContribut	0.36

图 5-9　PolkDot3D 参数界面

3. 完整代码

1）顶点着色器

```
uniform float SpecularContribution;
uniform vec3 LightPosition;
varying vec3 MCPosition;
varying float LightIntensity;

void main(void)
{
    float diffusecontribution  = 1.0 - SpecularContribution;
    vec3  ecPosition           = vec3(gl_ModelViewMatrix * gl_Vertex);
    vec3  tnorm                = normalize(gl_NormalMatrix * gl_Normal);
    vec3  lightVec             = normalize(LightPosition - ecPosition);
    vec3  reflectVec           = reflect(-lightVec, tnorm);
    vec3  viewVec              = normalize(-ecPosition);
    float diffuse              = max(dot(lightVec, tnorm), 0.0);
    float spec                 = 0.0;
    if(diffuse > 0.0)
      {
        spec = max(dot(reflectVec, viewVec), 0.0);
        spec = pow(spec, 16.0);
      }
    LightIntensity  = diffusecontribution * diffuse * 1.5 +
                      SpecularContribution * spec;
    MCPosition      = vec3 (gl_Vertex);
    gl_Position     = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

2）像素着色器

```
varying float LightIntensity;
varying vec3 MCPosition;
uniform vec3 Spacing;
uniform float DotSize;
uniform vec3 ModelColor, PolkaDotColor;

void main(void)
{
    float insidesphere, sphereradius, scaledpointlength;
    vec3 scaledpoint, finalcolor;
    scaledpoint       = MCPosition - (Spacing * floor(MCPosition/Spacing));
    scaledpoint       = scaledpoint - Spacing/2.0;
    scaledpointlength = length(scaledpoint);
```

```
        insidesphere        = step( scaledpointlength, DotSize) ;
        finalcolor          = vec3( mix( ModelColor, PolkaDotColor, insidesphere) ) ;
        gl_FragColor        = clamp( vec4( finalcolor * LightIntensity, 1.0) ,
                                    0.0, 1.0) ;
}
```

4. 效果图

如图 5-10 所示是 PolkDot3D 渲染的效果，第一行是各种不同的三维模型的渲染效果，第二行是同一只狗的三维模型在不同参数下的渲染效果。

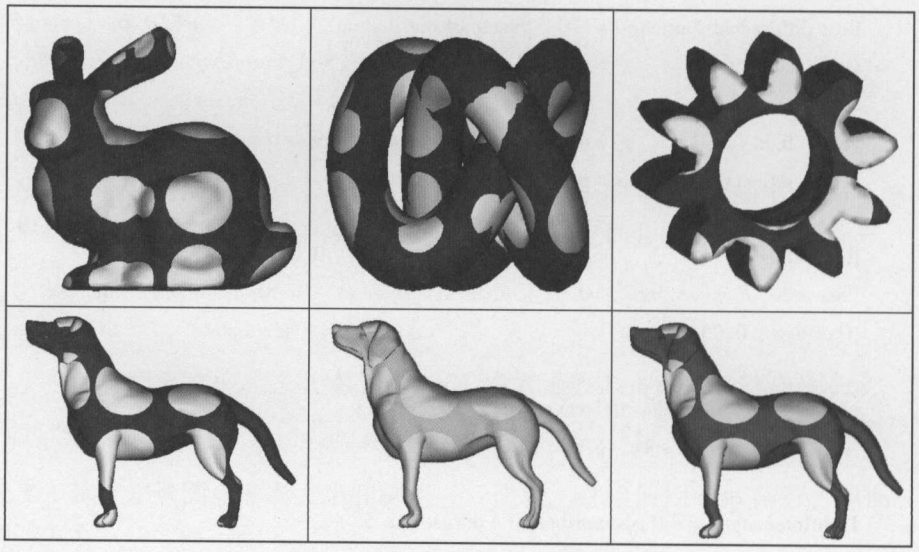


图 5-10　PolkDot3D 渲染效果

5.5　分形渲染

GLSL 语言不仅可以简单地计算光照来对三维模型进行渲染，而且还可以进行数学函数的计算，从而在三维模型表面显示由数学函数进行定义的某种有规律的图案。对于渲染来说，分形图案是一个具有特殊视觉效果的数学图案。分形图案是有规律的，放大后仍然和原来的图案相似。自然界中大部分形状不是有序的、稳定的，而是处于无序的、不稳定的和随机的状态中。例如，弯弯曲曲的海岸线、起伏不平的山脉、粗糙不堪的断面、变幻无常的浮云、令人眼花缭乱的满天繁星等。这些形状的特点都是极不规则或极不光滑的，这些形状都是分形图案。传统的欧几里得几何无法描述这一类复杂而又无规则的现象，需要用分形几何来描述。分形图案的特性是具有任意小尺度下的比例细节，也就是具有精细的结构。分形图案具有某种自相似形式，可能是近似的自相似或统计的自相似。分形可以用新的观念、新的手段透过无序的状态揭示在无序的混乱现象背后的规律。分形渲染就是要呈现一种不规则的、自相似的美，其目的不是要获得一种绘画或贴图效果，而是在 GPU 上执行一种通用的计算，实现分形图案数学函数的可视化。GPU 分形渲染是指通过 GLSL 程序把属于分形图形

集合的点显示出来，这样就可以在三维模型上绘制分形图案，从而使三维模型的渲染更加多彩多姿。

1. Mandelbrot 分形

Mandelbrot set 是数学家 Benoit Mandelbrot 研究的一个复数递归函数，其定义为：

$$Z_0 = c$$

$$Z_{n+1} = Z_n^2 + c$$

对于每一次迭代对 Z 的值求平方然后与一个不变的复数 c 相加。当选取某些特定的复数 c 值时，经过多次迭代后，函数值会达到无穷大，那么这些值不属于 Mandelbrot set。而选取其他复数 c 值不会使函数达到无穷大，这些值属于 Mandelbrot set。数学家 Mandelbrot 指出当 Z 的模大于 2 时，那么函数值就会达到无穷大，所以在编程中设定 Z 的平方大于 4 时停止迭代。但对于 Z 的平方小于 4 的区域，无论迭代多少次都不会使函数值达到无穷大，为了避免计算机进入死循环，这里需要设置允许迭代的最大次数，在超出最大迭代次数后就终止计算并使改点在函数内部，每一次迭代都为在 Mandelbrot set 函数外部的值指定一个颜色。

2. 算法步骤

1）顶点着色器

在顶点着色器中要进行光照强度计算，并设置绘制函数图形的坐标系，其中光照强度计算和之前的分析是一样的。

第一步：变量声明。

```
uniform vec3 LightPosition;
uniform float SpecularContribution;
uniform float DiffuseContribution;
uniform float Shininess;
varying float LightIntensity;
varying vec3  Position;
```

第二步：计算视角坐标。

```
vec3 ecPosition = vec3(gl_ModelViewMatrix * gl_Vertex);
```

第三步：计算顶点法向量。

```
vec3 tnorm      = normalize(gl_NormalMatrix * gl_Normal);
```

第四步：计算从渲染对象表面的当前点到光源位置的矢量。

```
vec3 lightVec   = normalize(LightPosition - ecPosition);
```

第五步：计算反射光线矢量。

```
vec3 reflectVec = reflect(-lightVec,tnorm);
```

第六步：计算查看位置方向上的矢量。

```
vec3 viewVec    = normalize(-ecPosition);
```

第七步：计算镜面反射、漫反射。

```
float spec          = max( dot( reflectVec, viewVec ), 0.0 );
spec                = pow( spec, Shininess );
```

第八步：计算光照强度。

```
LightIntensity    = DiffuseContribution *
                        max( dot( lightVec, tnorm ), 0.0 )  +
                        SpecularContribution * spec;
```

第九步：设置绘制图形的坐标系。这里假定纹理坐标 s、t 的值在[0,1]之间，为了更好地绘制 Mandelbrot set 图形，将纹理坐标 s、t 映射到[-2.5,2.5]之间。

```
Position            = vec3( gl_MultiTexCoord0 - 0.5 ) * 5.0;
```

第十步：计算顶点位置。

```
gl_Position         = ftransform( );
```

2）像素着色器

在像素着色器，这里主要完成颜色设置，包括 Mandelbrot Set 内部颜色 Inner Color 和外部颜色 outcolor1 和 outcolor2。内部颜色只使用一种颜色，外部颜色使用两种颜色，Mandelbrot Set 外部的值会根据迭代次数用 outcolor1 和 outcolor2 的线性差值得到。

第一步：变量声明。

```
varying vec3   Position;
varying float LightIntensity;
uniform float MaxIterations;
uniform float Zoom;
uniform float Xcenter;
uniform float Ycenter;
uniform vec3   InnerColor;
uniform vec3   OuterColor1;
uniform vec3   OuterColor2;
```

第二步：计算 Mandelbrot set 函数图形绘制的起使坐标。

```
float   real    = Position. x * Zoom + Xcenter;
float   imag    = Position. y * Zoom + Ycenter;
```

第三步：进入迭代运算，重新计算函数 Z 的值，直到跳出迭代循环。

```
for ( iter = 0.0; iter < MaxIterations && r2 < 4.0; ++ iter)
    {
        float tempreal = real;

        real = ( tempreal * tempreal) - ( imag * imag) + Creal;
```

```
        imag = 2.0 * tempreal * imag + Cimag;
        r2   = (real * real) + (imag * imag);
    }
```

第四步：计算 Mandelbrot set 函数内部值颜色。

```
if (r2 < 4.0)
    color = InnerColor;
```

第五步：根据迭代循环的次数，确定 Mandelbrot set 函数外部值的颜色。

```
color = mix(OuterColor1, OuterColor2, fract(iter * 0.05));
```

第六步：计算叠加光照强度后的颜色值。

```
color * = LightIntensity;
```

第七步：计算内置变量 gl_FragtColor 值，即最后显示的颜色。

```
gl_FragColor = vec4(color, 1.0);
```

3. 完整代码

1）顶点着色器

```
uniform vec3 LightPosition;
uniform float SpecularContribution;
uniform float DiffuseContribution;
uniform float Shininess;

varying float LightIntensity;
varying vec3  Position;

void main()
{
    vec3 ecPosition = vec3(gl_ModelViewMatrix * gl_Vertex);
    vec3 tnorm      = normalize(gl_NormalMatrix * gl_Normal);
    vec3 lightVec   = normalize(LightPosition - ecPosition);
    vec3 reflectVec = reflect(-lightVec, tnorm);
    vec3 viewVec    = normalize(-ecPosition);
    float spec      = max(dot(reflectVec, viewVec), 0.0);
    spec            = pow(spec, Shininess);
    LightIntensity  = DiffuseContribution *
                      max(dot(lightVec, tnorm), 0.0) +
                      SpecularContribution * spec;
    Position        = vec3(gl_MultiTexCoord0 - 0.5) * 5.0;
    gl_Position     = ftransform();

}
```

2）像素着色器

```
varying vec3  Position;
varying float LightIntensity;
uniform float MaxIterations;
uniform float Zoom;
uniform float Xcenter;
uniform float Ycenter;
uniform vec3  InnerColor;
uniform vec3  OuterColor1;
uniform vec3  OuterColor2;

void main()
{
    float  real  = Position.x * Zoom + Xcenter;
    float  imag  = Position.y * Zoom + Ycenter;
    float  Creal = real;
    float  Cimag = imag;
    float r2 = 0.0;
    float iter;
    for (iter = 0.0; iter < MaxIterations && r2 < 4.0; ++iter)
    {
        float tempreal = real;

        real = (tempreal * tempreal) - (imag * imag) + Creal;
        imag = 2.0 * tempreal * imag + Cimag;
        r2   = (real * real) + (imag * imag);
    }
    vec3 color;
    if (r2 < 4.0)
        color = InnerColor;
    else
        color = mix(OuterColor1, OuterColor2, fract(iter * 0.05));
    color *= LightIntensity;
    gl_FragColor = vec4(color, 1.0);
}
```

3）参数传递程序

```
public class GPURenderMandel : AbstractGPURender
{
    public GPURenderMandel()
    {
        VSFileName = "Data/Shaders/3dShader/Default/NPR/Mandel.vert";
```

```
            FSFileName = "Data/Shaders/3dShader/Default/NPR/Mandel.frag";
        }
        public override void PassParameter()
        {
            GL.Uniform3(GL.GetUniformLocation(this.handle,"LightPosition"),
                ConfigGPUCommon.Instance.LightPosition);
            GL.Uniform1(GL.GetUniformLocation(this.handle,"SpecularContribution"),
                ConfigGPUMandel.Instance.SpecularContribution);
            GL.Uniform1(GL.GetUniformLocation(this.handle,"DiffuseContribution"),
                ConfigGPUMandel.Instance.DiffuseContribution);
            GL.Uniform1(GL.GetUniformLocation(this.handle,"Shininess"),
                ConfigGPUMandel.Instance.Shininess);
            GL.Uniform1(GL.GetUniformLocation(this.handle,"MaxIterations"),
                ConfigGPUMandel.Instance.MaxIterations);
            GL.Uniform1(GL.GetUniformLocation(this.handle,"Zoom"),
                ConfigGPUMandel.Instance.Zoom);
            GL.Uniform1(GL.GetUniformLocation(this.handle,"Xcenter"),
                ConfigGPUMandel.Instance.Xcenter);
            GL.Uniform1(GL.GetUniformLocation(this.handle,"Ycenter"),
                ConfigGPUMandel.Instance.Ycenter);
            GL.Uniform3(GL.GetUniformLocation(this.handle,"InnerColor"),
                ConfigGPUMandel.Instance.InnerColor );
            GL.Uniform3(GL.GetUniformLocation(this.handle,"OuterColor1"),
                ConfigGPUMandel.Instance.OuterColor1 );
            GL.Uniform3(GL.GetUniformLocation(this.handle,"OuterColor2"),
                ConfigGPUMandel.Instance.OuterColor2 );
            GL.Uniform1(GL.GetUniformLocation(this.handle,"Creal"),
                ConfigGPUMandel.Instance.Creal);
            GL.Uniform1(GL.GetUniformLocation(this.handle,"Cimag"),
                ConfigGPUMandel.Instance.Cimag);
        }
    }
```

4. 界面设置

Mandelbrot 函数需要比较多的参数，如图 5-11 所示是其参数的界面，通过更改这些参数可以显示不同的分形结果。

Cimag	0
Creal	-1.5
DiffuseContribution	0.8
InnerColor	(1, 0, 0)
MaxIterations	50
OuterColor1	(0.4901961, 0, 1)
OuterColor2	(1, 1, 1)
Shininess	16
SpecularContribution	0.2
Xcenter	-0.0002
Ycenter	0.7383
Zoom	1

图 5-11　Mandelbrot 的参数界面

5. 效果图

实验一：如图 5-12 所示是各种三维模型在不同的分形参数设置下的显示效果，从中可以看出输入的参数不同，分形的图案也不一样，但分形的局部进行放大后还是得到和原来的图案相似的形状。

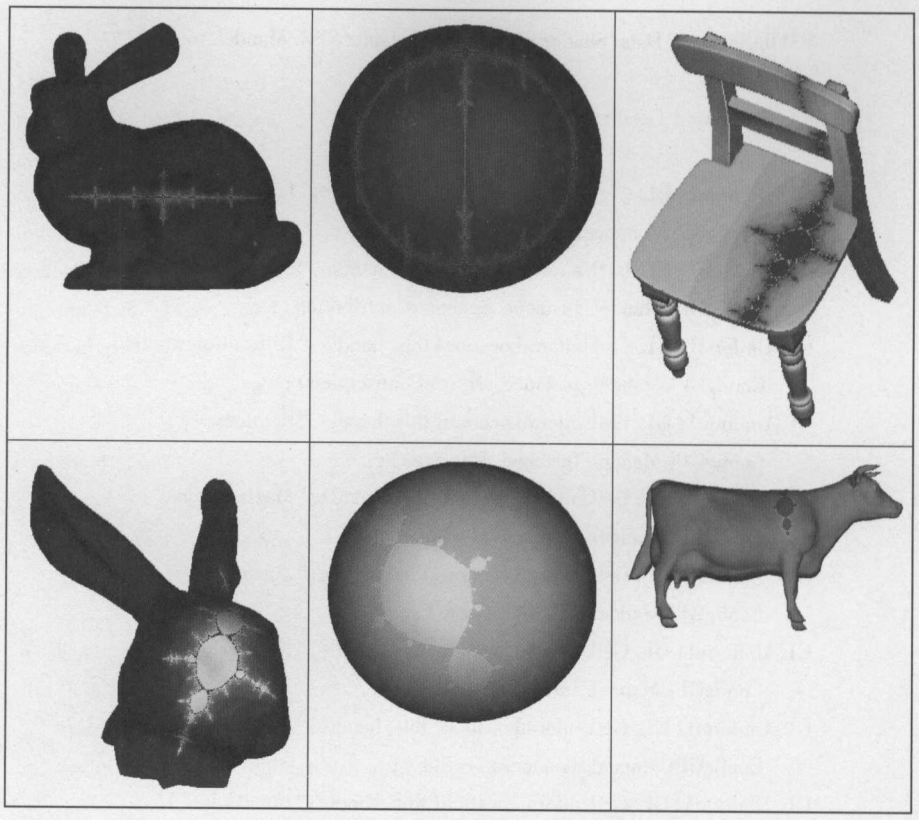


图 5-12　Mandelbrot 分形效果图

实验二：Mandelbrot 分形是由分形公式里的参数决定的，不同的参数图案不同，如图 5-13 所示为某些特定的参数得到的比较明显的图案结果。第二行、第三行是放大后的图案效果，从中可以看出，放大后的图案展现出和原来图案整体相似的形状。

6. Julia set 分形渲染

Julia set 是 Mandelbrot set 的一个特例，Mandelbrot set 中的每一个点都可以用来生成一个 Julia set。Julia set 渲染要求把 $Z^2 + c$ 中的初始值复数 c 初始化为 Mandelbrot set 中的一个点的值。

Julia set 顶点着色器和 Mandelbrot set 着色器代码是一样的。Julia set 的顶点着色器主要是给初始的复数一个值，而且这个值是 Mandelbrot set 分形函数上的一个点。为了获得不同的渲染效果，即指定一个初始值就获得一种渲染效果，所以要将这个初始值设为 Uniform 变量，以便可以通过应用程序界面随时更改初始值。

7. 效果图

如图 5-14 所示是把参数设置为不同设置时得到的 Julia set 分形渲染效果。从中可以看出不同的分形图案随着参数的变化而变化。

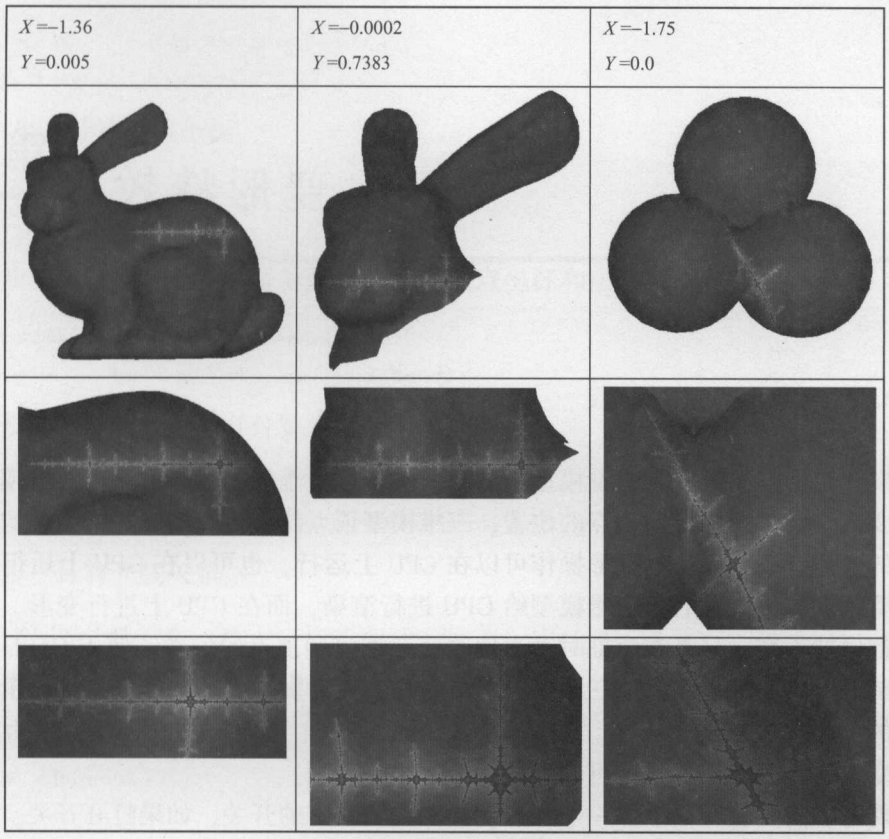


图 5-13　Mandelbrot 分形效果图

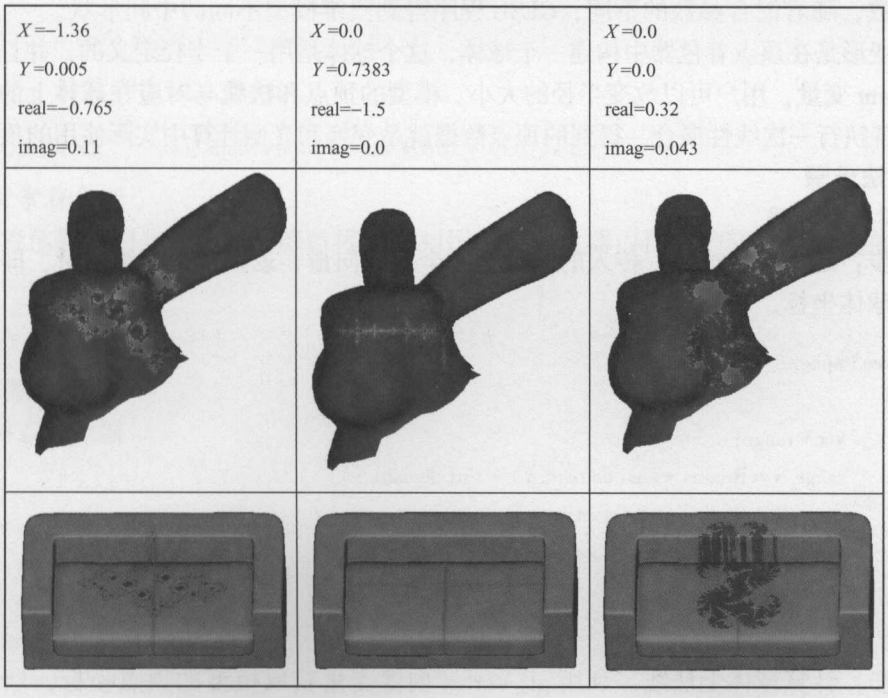


图 5-14　Jualia set 分形渲染

第6章 变形特效

6.1 球形变形特效

三维模型变形是指把一个三维模型随着时间的变化，慢慢地变为另一个三维模型。三维模型变形要改变的是三维模型顶点的位置，三维模型顶点位置改变后，三维模型的形状就发生了相应的变化。三维模型的变形操作可以在 CPU 上运行，也可以在 GPU 上运行。在 CPU 上运行需要每次传递变形后的三维模型给 GPU 进行渲染，而在 GPU 上进行变形，每次三维模型形状都保持不变，只是在渲染时动态地改变三维模型顶点的位置。使用 GLSL 语言可以动态地控制三维模型的形状，使三维模型的形状在渲染时发生变形，从而最终显示的三维模型和原来的三维模型外观不一样。根据输入的时间参数，可以控制三维模型发生变形的程度，如果变形是连续的，那么就可以生成一个完整的动画。

在实现时，需要在界面上加一个开启动画和关闭动画的开关，如果打开开关，则开启计时器，计时器根据固定的时间间隔更新，每次更新就重新渲染一次，而每次渲染都会计算新的混合参数，随着混合参数的不同，GLSL 程序得到三维模型不同的中间形状。

球形变形是在顶点着色器中构建一个球体，这个球体是用一个半径定义的，并且半径设为外部 Uniform 变量，用户可以改变半径的大小。模型的顶点和法线与对应在球体上的顶点和法线点之间各执行一次线性混合，得到的顶点数据就是变换和光照计算中实际使用的顶点数据。

1. 算法步骤

1）顶点着色器

第一步：定义一个球体。传入的参数是一个二维向量，返回值是三维向量，即有平面坐标转换成球体坐标。

```
vec3 sphere( vec2 domain)
{
    vec3 range;
    range. x = Radius * cos( domain. y) * sin( domain. x) ;
    range. y = Radius * sin( domain. y) * sin( domain. x) ;
    range. z = Radius * cos( domain. x) ;
    return range;
}
```

第二步：计算球体上法线。利用 gl_Vertex 内置变量获取模型的顶点坐标，然后利用前面定义的球体，计算球体上的法线向量。

```
vec2 p0          = gl_Vertex. xy * TWO_PI;
vec3 normal      = sphere( p0) ;
```

第三步：计算球体坐标。

```
vec3 r0          = Radius * normal;
vec3 vertex      = r0;
```

第四步：执行线性差值，计算变换用到的顶点坐标和法线。

```
normal      = mix( gl_Normal, normal, Blend) ;
vertex      = mix( gl_Vertex. xyz, vertex, Blend) ;
```

第五步：归一化法线，并计算眼睛坐标。

```
normal           = normalize( gl_NormalMatrix * normal) ;
vec3 position    = vec3( gl_ModelViewMatrix * vec4( vertex, 1. 0) ) ;
```

第六步：计算光线矢量。

```
vec3 lightVec = normalize( LightPosition - position) ;
```

第七步：根据法线和光线矢量计算漫反射强度。

```
float diffuse = max( dot( lightVec, normal) , 0. 0) ;
if ( diffuse < 0. 125)
    diffuse = 0. 125;
```

第八步：计算最终着色值。

```
Color = vec4( SurfaceColor * diffuse, 1. 0) ;
```

第九步：计算齐次顶点坐标。

```
gl_Position      = gl_ModelViewProjectionMatrix * vec4( vertex, 1. 0) ;
```

2）像素着色器

像素着色器使用非常简单的着色模式，利用顶点着色器内计算的颜色值，直接复制给像素着色器即可。

```
gl_FragColor      = Color;
```

2. 完整代码

1）顶点着色器

```
varying vec4 Color;
uniform vec3 LightPosition;
uniform vec3 SurfaceColor;
const float twopi = 6. 28318;
const float pi = 3. 14159;
uniform float radius;
uniform float blend;
```

```
vec3 sphere(vec2 domain)
{
    vec3 range;
    range.x = radius * cos(domain.y) * sin(domain.x);
    range.y = radius * sin(domain.y) * sin(domain.x);
    range.z = radius * cos(domain.x);
    return range;
}

void main(void)
{
    vec2 p0 = gl_Vertex.xy * twopi;
    vec3 normal = sphere(p0);;
    vec3 r0 = radius * normal;
    vec3 vertex = r0;
    normal = normal * blend + gl_Normal * (1.0 - blend);
    vertex = vertex * blend + gl_Vertex.xyz * (1.0 - blend);
    normal = normalize(gl_NormalMatrix * normal);
    vec3 position = vec3(gl_ModelViewMatrix * vec4(vertex,1.0));
    vec3 lightVec    = normalize(LightPosition - position);
    float diffuse    = max(dot(lightVec,normal),0.0);
    if (diffuse < 0.125)
        diffuse = 0.125;
    Color = vec4(SurfaceColor * diffuse,1.0);
    gl_Position = gl_ModelViewProjectionMatrix * vec4(vertex,1.0);
}
```

2）像素着色器

```
varying vec4 Color;
void main (void)
{
    gl_FragColor = Color;
}
```

3）参数传递

球形变形动画效果参数传递代码如下。其中动画的生成是通过改变参数 Blender 来控制的，随着 Blender 参数的变化，生成不同的中间形状。当 Blender 为 0 时，显示原始三维模型，当 Blender 为 1 时显示球形。

```
public class GPURenderSphereMorph:AbstractGPURender
{
    public GPURenderSphereMorph()
```

```
    {
        VSFileName = "GLSL/Animation/SphereMorph.vert";
        FSFileName = "GLSL/Animation/SphereMorph.frag";
    }

    public override void PassParameter()
    {
        if (ConfigGPUCommon.Instance.EnableTime)
        {
            ConfigGPUSphereMorph.Instance.Blend =
                (ConfigGPUSphereMorph.Instance.Blend + 0.001f) % 1;
        }
        GL.Uniform3(GL.GetUniformLocation(this.handle, "LightPosition"),
            ConfigGPUCommon.Instance.LightPosition);
        GL.Uniform3(GL.GetUniformLocation(this.handle, "SurfaceColor"),
            ConfigGPUSphereMorph.Instance.SurfaceColor);
        GL.Uniform1(GL.GetUniformLocation(this.handle, "radius"),
            ConfigGPUSphereMorph.Instance.Radius);
        GL.Uniform1(GL.GetUniformLocation(this.handle, "blend"),
            ConfigGPUSphereMorph.Instance.Blend);
        }
    }
```

3. 效果图

实验一：如图6-1所示是兔子头三维模型变为一个球形的几个中间过程，从中可以看出随着时间的变化，兔子头慢慢地向球形进行变形。

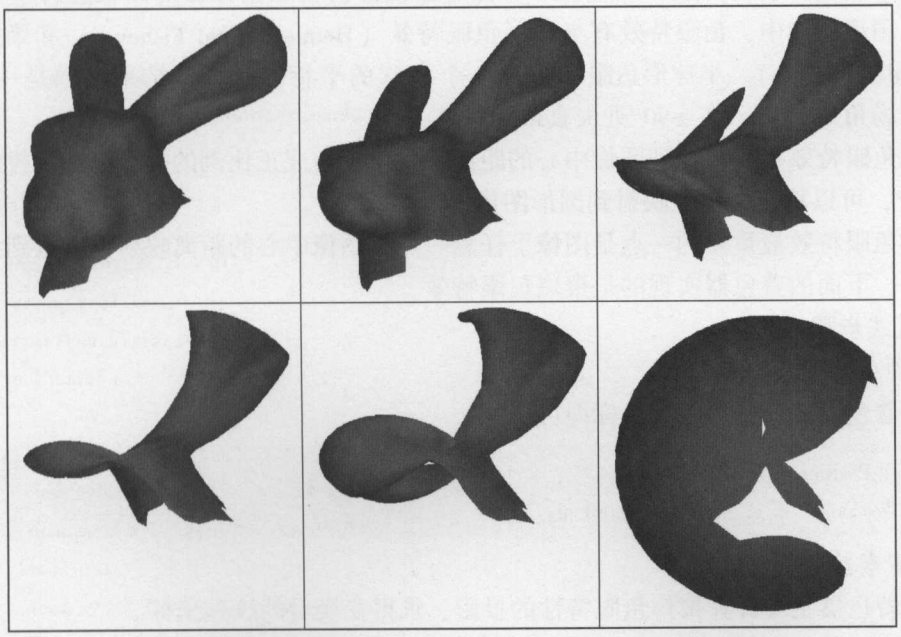


图6-1　兔子头三维模型变形效果

实验二：如图 6-2 所示是牛三维模型变为一个球形的几个中间过程，从中可以看出随着时间的变化，牛慢慢地向球形进行变形。但这种变形方法是一种简单、直接、基本的方法，得到中间形状不是很理想，有些中间形状既不像一个球，也不像牛。

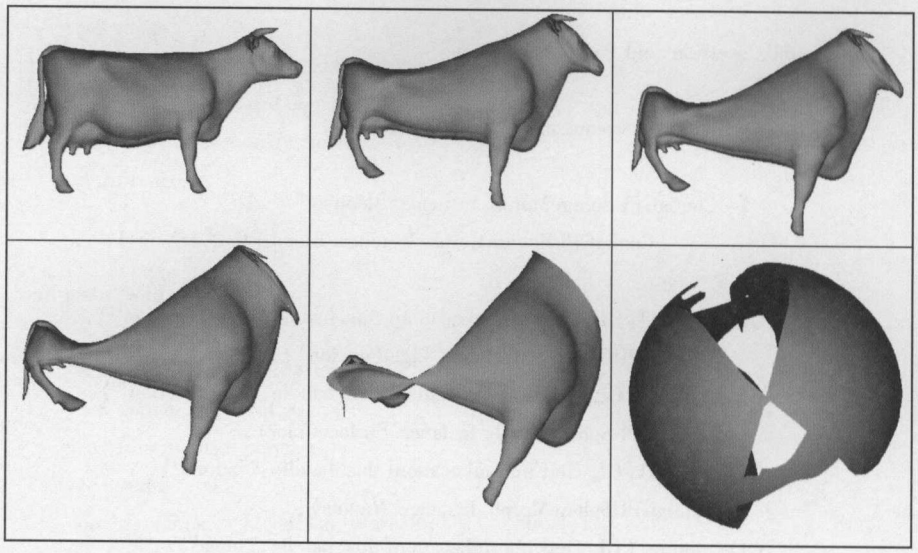


图 6-2　牛三维模型变形效果

6.2　鱼眼特效

鱼眼特效可以制造图像扭曲的效果，鱼眼效果在图像的边角处对图像内容进行了扭曲，看上去就像通过鱼眼透镜显示一般（尽管任意透镜都会在某些方面对图像进行一些扭曲）。在计算机图形渲染中，鱼眼特效有半球形鱼眼特效（Hemispherical Fisheye）和聚焦鱼眼特效（Angular Fisheye）。半球形鱼眼特效是一个半球的平行投影，其结果图像是一个圆形，最宽的视场角为 180°，在 ±90°处失真最严重。

聚焦鱼眼特效是投影面到图像中心的距离与观察视角成正比例的一个投影，投影范围可达到 360°，可以将整个环境映射到圆形图像上。

聚焦鱼眼特效最重要的一点是图像上任意一点到图像中心的距离映射为该点到投影球体投影角度。下面的着色器实现的是聚焦鱼眼特效。

1. 算法步骤

1）顶点着色器

顶点着色器只需传入纹理坐标即可。

```
gl_Position = ftransform( );
TexCoord    = gl_MultiTexCoord0. st;
```

2）像素着色器

像素着色器主要计算聚焦鱼眼特性的投影，求得变换后的纹理坐标。

第一步：归一化纹理坐标，即将图像的纹理坐标规范到（-1,1）之间。

```
vec2 xy = 2.0 * TexCoord.xy - 1.0;
```

第二步：计算归一化后的矢量长度。

```
float d = length(xy);
```

第三步：如果求得纹理坐标矢量长度在投影范围内则进行投影变换。

```
d = length(xy * maxFactor);
float z = sqrt(1.0 - d * d);
float r = atan(d,z)/PI;
float phi = atan(xy.y,xy.x);
uv.x = r * cos(phi) + 0.5;
uv.y = r * sin(phi) + 0.5;
```

第四步：如果求得的纹理坐标矢量长度不在投影范围内则不进行处理。

```
uv = TexCoord.xy;
```

第五步：根据纹理坐标获取纹理值。

```
vec4 c = texture2D(tex0,uv);
```

第六步：计算最终的输出颜色值。

```
gl_FragColor = c;
```

2. 完整代码

1）顶点着色器

```
varying vec2 TexCoord;
void main()
{
    gl_Position = ftransform();
    TexCoord    = gl_MultiTexCoord0.st;
}
```

2）像素着色器

```
uniform sampler2D tex0;
varying vec2 TexCoord;
const float PI = 3.1415926535;
void main()
{
  float aperture = 178.0;
  float apertureHalf = 0.5 * aperture * (PI / 180.0);
  float maxFactor = sin(apertureHalf);
  vec2 uv;
  vec2 xy = 2.0 * TexCoord.xy - 1.0;
  float d = length(xy);
```

```
    if (d < (2.0 - maxFactor))
    {
        d = length(xy * maxFactor);
        float z = sqrt(1.0 - d * d);
        float r = atan(d,z) / PI;
        float phi = atan(xy.y,xy.x);
        uv.x = r * cos(phi) + 0.5;
        uv.y = r * sin(phi) + 0.5;
    }
    else
    {
        uv = TexCoord.xy;
    }
    vec4 c = texture2D(tex0,uv);
    gl_FragColor = c;
}
```

第 7 章 噪声渲染

7.1 柏林噪声

计算机在渲染三维模型时，可以很精确地设置三维模型的颜色，如设置为黄色、红色等。但现实世界的三维模型通常具有不完美的颜色，如一个比较旧的瓷碗、有木纹的椅子、蓝天上的白云等。只有加入了不完美的效果，三维模型的渲染才能够看起来更加真实。虽然陈旧的瓷碗上具有一些斑点，这些斑点是随机出现的。如果简单地放置几个斑点，那么渲染效果看起来仍旧很假。目前解决这种随机出现的、没有规律的、视觉上看起来逼真的技术称为基于噪声的渲染。这是由 Perlin 首先提出来的一种算法，通常称为柏林噪声。除了柏林噪声之外，还有其他的噪声算法。

柏林噪声和日常的噪声不一样，日常的噪声随着时间变化，得到的结果不一样，完全是随机的。而柏林噪声在输入一样的情况下，得到的值都是一样的。只是结果展现出一种视觉上看起来没有规律的效果。这样的话就可以在渲染时，为三维模型增加真实感。假如输入一样，每次柏林噪声得到的结果不一样，那么就无法保证一个三维模型每次渲染的结果一样。柏林噪声可以用在各种三维特效的渲染中。例如，云、火、烟、风等大自然现象；大理石、木头、山等自然物质；石灰、沥青、水泥等人造材料；斑点、灰尘、锈迹、脏污等缺陷；褶皱、凸起、颜色起伏等特征；或者还可以用在动作上，如悸动、跳跃等。这些都是没有规律的现象，通过柏林噪声可以进行模拟。

1. 柏林噪声原理

创建柏林噪声需要两个函数：一个是噪声函数；另一个是插值函数。一个噪声函数基本上是一个种子随机发生器，它需要一个整数作为参数，然后根据这个参数返回一个随机数(一般为 0 ~ 1)。如果两次都传同一个参数进来，它就会产生两次相同的数，这条规律非常重要，否则柏林函数只是生成一堆垃圾。如图 7-1 所示是一个噪声函数的图示。

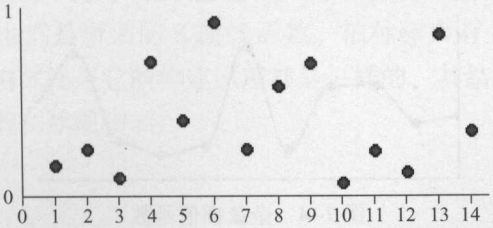


图 7-1 噪声函数

噪声函数产生的数据仅是一些离散的整数数据点，当输入非整数时就需要插值计算输出值。通过在整数值之间平滑地插值，可以定义一个带有一个非整参数的连续函数，这个函数称为插值函数。如图 7-2 所示是一个平滑插值函数的图示。

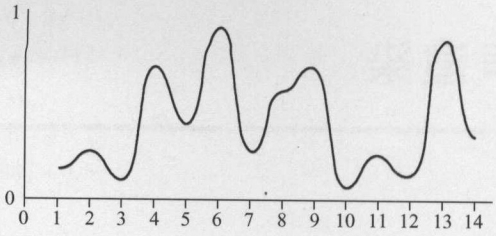


图 7-2　平滑插值函数

插值函数有很多种，常用的有线性插值、余弦插值和立方插值。线性插值是最简单的插值函数，即根据已知的两点坐标求中间某点坐标时采用一个线性组合，代码如下。

```
function Linear_Interpolate(a,b,x)
    return a * (1 - x) + b * x
end of function
```

线性插值示例图见图 7-3。

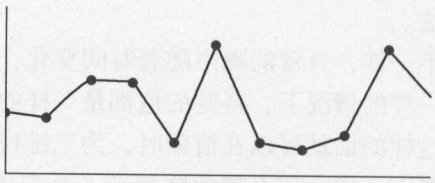


图 7-3　线性插值函数

其中随机数是[0,1]，可以看到，线性插值生成的曲线都是分段的直线。

余弦插值是根据已知的两点坐标求中间某点坐标时采用余弦公式，代码如下。

```
function Cosine_Interpolate(a,b,x)
    ft = x * 3.1415926
    f = (1 - cos(ft) * 0.5)
    return a * (1 - f) + b * f
end of function
```

线性插值示例图见图 7-4。

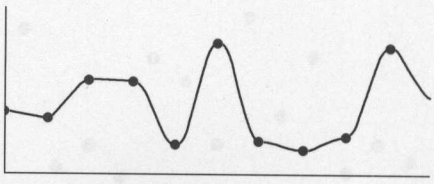


图 7-4　余弦插值函数

相对来讲，它插值的结果比较光滑。

立方插值也称三次插值，产生的曲线很光滑，但要以牺牲速度为代价。该插值函数接收5个参数，包括4个顶点数据，1个 x 值，代码如下。

```
v0 = the point before a
v1 = the point a
v2 = the point b
v3 = the point after b
function Cubic_Interpolate(v0,v1,v2,v3,x)
    P = (v3 - v2) - (v0 - v1)
    Q = (v0 - v1) - P
    R = v2 - v0
    S = v1
    return Px3 + Qx2 + Rx + S
end of function
```

立方插值函数取值见图7-5，立方插值函数见图7-6。

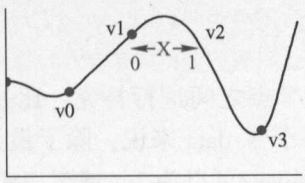


图7-5 立方插值函数取值

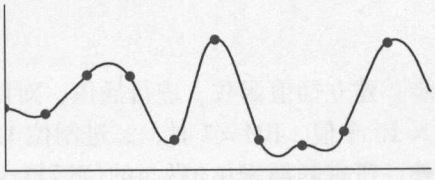


图7-6 立方插值函数

第一种插值方式最快，但平滑度差，第三种插值方式平滑度最好，但速度最慢，第二种插值方式在速度和平滑度上介于两者之间。线性插值的方法保证了插值函数的输出在插值点上取得了正确的结果，但不能保证插值函数在插值点上的导数值。线性插值生成的纹理有褶皱瑕疵，也就是界线太明显，图像显得不自然。为了解决这个问题需要在线性插值的基础上做一些改进。

埃尔米特插值是另一类插值问题，这类插值在给定的节点处，不但要求插值多项式的函数值与被插函数的函数值相同。同时还要求在节点处，插值多项式的一阶直至指定阶的导数值，也与被插函数的相应阶导数值相等，这样的插值称为埃尔米特插值，或称 Hermite 插值。

利用埃尔米插值可以在保证函数输出的基础上保证插值函数的导数在插值点上为0，这样就提供了平滑性。插值点为0、1，结果值为0、1，导数为0、0，则可以求得埃尔米插值函数为 $s(x) = -2x^3 + 3x^2$ 也就是所谓的S曲线函数。柏林噪声有一维、二维、三维或更高维的形式，不管是几维的柏林噪声它的构建原理都是一样的，其结果对应为一维、二维、三维或更高维的数据，可存放在数组中。

1. 算法步骤

1）一维柏林噪声

第一步：创建噪声函数。噪声函数就是一个生成随机数的函数，生成一维的值域位 -1 ~ 1 的随机函数。

```
private void initNoise( )
{
    int i,j,k ;
    Random ra = new Random(30757);
    g1[i] = (double)((ra. Next( ) % (B + B)) - B) / B;
   while ((--i) >=1)
   {
      k = p[i];
      p[i] = p[j = ra. Next( ) % B];
      p[j] = k;
    }
    for (i =0;i < B +2;i ++)//i 由 0 -5
   {
      p[B +i] = p[i];
      g1[B +i] = g1[i];
   }
}
```

第二步：建立插值函数。进行插值，对随机函数的两个点之间进行补完，比如在 0 - 1 之间再插入 10 个值。BM = 7 时，二进制位 111，显然，对于 N. data 来说，除了最开始的一位为 1 以外，所有的都为 0，做与的位运算永远是 0，直到 vec 可以为 1，或者 BM 足够大，达到了 8191 为止。

```
private void setup(double vec)
{
    double t;
    t = vec + (int)N. date;
    b0 = ((int)t) & BM;
    b1 = (b0 +1) & BM;
    r0 = t - (int)t;
    r1 = (double)(r0 -1);
}
private double lerp(double t,double a,double b)
{ return (a +t * (b - a));}
```

第三步：建立埃尔米插值函数，改进插值函数得到更为平滑的噪声数据。

```
private double sCurve(double t)
{
    double ret;
    ret = (double)(t * t * (3 -2 * t));
    return ret;
}
```

第四步：根据噪声函数、插值函数及埃尔米插值函数建立 Coherent Noise 噪声。第一次

vec 为 0.0625，第一次 b0 = 0，b1 = 1，r0 = 0.0625，r1 = -0.9375。

```
private double noise1(double arg)
 {
      int bx0,bx1;
      double rx0,rx1,sx,u,v;
      double vec;
      vec = arg;
      if (start == 1)
      {
          start = 0;
              initNoise();
      }
      setup(vec);
      bx0 = b0;bx1 = b1;rx0 = r0;rx1 = r1;
      sx = sCurve(rx0);
      u = rx0 * g1[p[bx0]];
      v = rx1 * g1[p[bx1]];
      return (lerp(sx,u,v));
 }
```

第五步：叠加 Coherent Noise 噪声，建立柏林噪声。进一步对噪声函数进行叠加，不过效率会大大降低。

```
private double PerlinNoise1D(double x,double alpha,double beta,int n)
 {
     int i;
     double val,sum = 0;
     double p,scale = 1;
     p = x;
     for (i = 0;i < n;i ++)
     {
         val = noise1(p);
         sum + = val / scale;
         scale * = alpha;
         p * = beta;
      }
     return (sum);
 }
```

以上是建立一维柏林噪声的步骤，对于二维和三维柏林噪声的建立思路和一维噪声是一样的，不同之处在于各个函数传入的参数不一样。

2）二维柏林噪声

第一步：建立噪声函数。

```
private void initNoise()
{
    int i,j,k ;
    Random ra = new Random(30757);
    for (i=0;i<B;i++)
    {
        p[i] =i;//0-3
        for (j=0;j<2;j++)
        g2[i,j] = (double)((ra.Next() % (B+B)) -B) / B;
        normalize2(g2[i,0],g2[i,1]);
    }
  while ((--i) >=1)
  {
      k=p[i];
      p[i] =p[j=ra.Next() % B];
      p[j] =k;
   }
   for (i=0;i<B+2;i++)//i由0到5
  {
      p[B+i] =p[i];
      for (j=0;j<2;j++)
      g2[B+i,j] =g2[i,j];
  }
}
```

第二步：建立插值函数。插值方法和一维是一样的。

```
private void setup(double vec)
{
    double t;
    t = vec + (int)N.date;
    b0 = ((int)t) & BM;
    b1 = (b0+1) & BM;
    r0 = t - (int)t;
    r1 = (double)(r0-1);
}
private double lerp(double t,double a,double b)
{ return (a+t*(b-a));}
```

第三步：建立埃尔米插值函数，改进插值函数得到更为平滑的噪声数据。

```
private double sCurve(double t)
{
    double ret;
```

```
        ret = (double)(t * t * (3 - 2 * t));
        return ret;
    }
```

第四步：根据噪声函数、插值函数及埃尔米插值函数建立 Coherent Noise 噪声。

```
    private double noise2(double vec0,double vec1)
    {
        int bx0,bx1,by0,by1,b00,b10,b01,b11;
        double rx0,rx1,ry0,ry1,sx,sy,a,b,u,v;
        double[] q = new double[2];
        int i,j;
        if (start == 1)
        {
            start = 0;
            initNoise();
        }
        setup(vec0);
        bx0 = b0;bx1 = b1;rx0 = r0;rx1 = r1;
        setup(vec1);
        by0 = b0;by1 = b1;ry0 = r0;ry1 = r1;
        i = p[bx0];
        j = p[bx1];
        b00 = p[i + by0];
        b10 = p[j + by0];
        b01 = p[i + by1];
        b11 = p[j + by1];
        sx = sCurve(rx0);
        sy = sCurve(ry0);
        q[0] = g2[b00,0];q[1] = g2[b00,1];u = at2(rx0,ry0,q[0],q[1]);
        q[0] = g2[b10,0];q[1] = g2[b10,1];v = at2(rx1,ry0,q[0],q[1]);
        a = lerp(sx,u,v);
        q[0] = g2[b01,0];q[1] = g2[b01,1];u = at2(rx0,ry1,q[0],q[1]);
        q[0] = g2[b11,0];q[1] = g2[b11,1];v = at2(rx1,ry1,q[0],q[1]);
        b = lerp(sx,u,v);
        return lerp(sy,a,b);
    }
```

第五步：叠加 Coherent Noise 噪声，建立二维柏林噪声。

```
    private double PerlinNoise2D(double x,double y,double alpha,double beta,int n)
    {
        int i;
        double val,sum = 0;
```

```
        double[ ] p = new double[2];
        double scale = 1;
        p[0] = x;
        p[1] = y;
        for (i = 0;i < n;i ++)
        {
          val = noise2(x,y);
          sum + = val / scale;
          scale * = alpha;
          p[0] * = beta;
          p[1] * = beta;
        }
        return (sum);
    }
```

3）三维柏林噪声

第一步：建立噪声函数。

```
    private void initNoise( )
    {
        int i,j,k ;
        Random ra = new Random(30757);
        for (i = 0;i < B;i ++)
        {
            p[i] = i;//0 - 3
            for (j = 0;j < 3;j ++)
            {
                g3[i,j] = (double)((ra. Next( ) % (B + B)) - B) / B;
                normalize3(g3[i,0],g3[i,1],g3[i,2]);
            }
          while ((--i) >= 1)
          {
            k = p[i];
            p[i] = p[j = ra. Next( ) % B];
            p[j] = k;
          }
          for (i = 0;i < B + 2;i ++)//i 由 0 - 5
        {
            p[B + i] = p[i];
            for (j = 0;j < 3;j ++)
            g3[B + i,j] = g3[i,j];
        }
    }
```

第二步：建立插值函数。

```
private void setup(double vec)
{
    double t;
    t = vec + (int)N. date;
    b0 = ((int)t) & BM;
    b1 = (b0 + 1) & BM;
    r0 = t - (int)t;
    r1 = (double)(r0 - 1);
}
private double lerp(double t,double a,double b)
{ return (a + t * (b - a));}
```

第三步：建立埃尔米插值函数，改进插值函数得到更为平滑的噪声数据。

```
private double sCurve(double t)
{
    double ret;
    ret = (double)(t * t * (3 - 2 * t));
    return ret;
}
```

第四步：根据噪声函数、插值函数及埃尔米插值函数建立 Coherent Noise 噪声。下面的三维柏林噪声完整代码会有 Coherent Noise 噪声代码。

第五步：叠加 Coherent Noise 噪声，建立三维柏林噪声。

```
private double PerlinNoise3D(double x,double y,double z,double alpha,double beta,int n)
{
    int i;
    double val,sum = 0;
    double[ ] p = new double[3];
     double scale = 1;
     p[0] = x;
     p[1] = y;
     p[2] = z;
     for (i = 0;i < n;i ++)
     {
         val = noise3(p[0],p[1],p[2]);
         sum + = val / scale;
         scale * = alpha;
         p[0] * = beta;
         p[1] * = beta;
         p[2] * = beta;
     }
```

```
    return (sum);
}
```

三维柏林噪声完整代码如下

下面是通过柏林噪声生成三维纹理贴图的相关函数。在生成三维纹理贴图之后，GLSL就可以使用贴图得到相应的渲染效果。

（1）生成三维柏林噪声数据。

```
private double noise3(double vec0,double vec1,double vec2)
{
    int bx0,bx1,by0,by1,bz0,bz1,b00,b10,b01,b11;
    double rx0,rx1,ry0,ry1,rz0,rz1,sy,sz,a,b,c,d,t,u,v;
    double[] q = new double[3];
    int i,j;
    if (start == 1)
    {
        start = 0;
        initNoise();
    }
    setup(vec0);
    bx0 = b0;bx1 = b1;rx0 = r0;rx1 = r1;
    setup(vec1);
    by0 = b0;by1 = b1;ry0 = r0;ry1 = r1;
    setup(vec2);
    bz0 = b0;bz1 = b1;rz0 = r0;rz1 = r1;
    i = p[bx0];
    j = p[bx1];
    b00 = p[i + by0];
    b10 = p[j + by0];
    b01 = p[i + by1];
    b11 = p[j + by1];
    t = sCurve(rx0);
    sy = sCurve(ry0);
    sz = sCurve(rz0);
    q[0] = g3[b00 + bz0,0];
    q[1] = g3[b00 + bz0,1];
    q[2] = g3[b00 + bz0,2];
    u = at3(rx0,ry0,rz0,q[0],q[1],q[2]);
    q[0] = g3[b10 + bz0,0];
    q[1] = g3[b10 + bz0,1];
    q[2] = g3[b10 + bz0,2];
    v = at3(rx1,ry0,rz0,q[0],q[1],q[2]);
    a = lerp(t,u,v);
```

```
        q[0] = g3[b01 + bz0,0];
        q[1] = g3[b01 + bz0,1];
        q[2] = g3[b01 + bz0,2];

        u = at3(rx0,ry1,rz0,q[0],q[1],q[2]);
        q[0] = g3[b11 + bz0,0];
        q[1] = g3[b11 + bz0,1];
        q[2] = g3[b11 + bz0,2];
        v = at3(rx1,ry1,rz0,q[0],q[1],q[2]);
        b = lerp(t,u,v);
        c = lerp(sy,a,b);
        q[0] = g3[b00 + bz1,0];
        q[1] = g3[b00 + bz1,1];
        q[2] = g3[b00 + bz1,2];
        u = at3(rx0,ry0,rz1,q[0],q[1],q[2]);
        q[0] = g3[b10 + bz1,0];
        q[1] = g3[b10 + bz1,1];
        q[2] = g3[b10 + bz1,2];
        v = at3(rx1,ry0,rz1,q[0],q[1],q[2]);
        a = lerp(t,u,v);
        q[0] = g3[b01 + bz1,0];
        q[1] = g3[b01 + bz1,1];
        q[2] = g3[b01 + bz1,2];
        u = at3(rx0,ry1,rz1,q[0],q[1],q[2]);
        q[0] = g3[b11 + bz1,0];
        q[1] = g3[b11 + bz1,1];
        q[2] = g3[b11 + bz1,2];
        v = at3(rx1,ry1,rz1,q[0],q[1],q[2]);
        b = lerp(t,u,v);
        d = lerp(sy,a,b);
        return lerp(sz,c,d);
    }
```

(2) 生成噪声贴图数组。

```
    public void make3DNoiseTexture()
    {
        int f,i,j,k,inc;
        int numOctaves = 4;
        double[ ]ni = new double[3];
        double inci,incj,inck;
        int frequency = 4;
        int ptrnum = 0;
```

```
        double amp = 0.5;
        double persistence = 0.5;
        if( ConfigGPUCommon. Instance. NoiseExist)
            persistence = ConfigGPUCommon. Instance. NoisePersistence;

        for( f = 0, inc = 0; f < numOctaves;
            ++ f, frequency = frequency * 2, ++ inc, amp * = persistence)
        {
            SetNoiseFrequency( frequency) ;
            ptrnum = 0;
            ni[0] = ni[1] = ni[2] = 0;
            inci = 1.0/( Noise3DTexSize/frequency) ;
            for( i = 0; i < Noise3DTexSize; ++ i, ni[0] += inci)
            {
                incj = 1.0/( Noise3DTexSize/frequency) ;
                for( j = 0; j < Noise3DTexSize; ++ j, ni[1] += incj)
                {
                    inck = 1.0/( Noise3DTexSize/frequency) ;
                    for( k = 0; k < Noise3DTexSize;
                        ++ k, ni[2] += inck, ptrnum += 4)
                    {
                      Noise3DTexPtr[ ptrnum + inc] =
                      (byte)( (( noise3( ni[0], ni[1], ni[2]) + 1) * amp) * 64) ;
                    }
                }
            }

        }
    }
```

（3）生成三维纹理贴图。

这个函数通过柏林噪声算法得到的数据来生成 OpenGL 的三维纹理。

```
public void init3DNoiseTexture( int texSize)
    {
        GL. BindTexture( TextureTarget. Texture3D, 0) ;
        GL. TexParameter( TextureTarget. Texture3D,
            TextureParameterName. TextureWrapS,
            ( int) TextureWrapMode. Repeat) ;
        GL. TexParameter( TextureTarget. Texture3D,
            TextureParameterName. TextureWrapT,
            ( int) TextureWrapMode. Repeat) ;
        GL. TexParameter( TextureTarget. Texture3D,
```

```
                TextureParameterName. TextureWrapR,
                (int)TextureWrapMode. Repeat);
            GL. TexParameter(TextureTarget. Texture3D,
                TextureParameterName. TextureMinFilter,
                (int)TextureMinFilter. Linear);
            GL. TexParameter(TextureTarget. Texture3D,
                TextureParameterName. TextureMagFilter,
                (int)TextureMagFilter. Linear);
            IntPtr p = Marshal. AllocHGlobal(64 * 64 * 64 * 4);
            Marshal. Copy(Noise3DTexPtr,0,p,64 * 64 * 64 * 4);
            GL. TexImage3D(TextureTarget. Texture3D,0,
                PixelInternalFormat. Rgba,
                texSize,texSize,texSize,0,
                PixelFormat. Rgba,PixelType. Byte,p);
        }
```

7.2　自然材质渲染

在通过柏林噪声生成了 OpenGL 三维纹理贴图之后，就可以在 GLSL 语言中使用这些贴图数据从而得到云、火等各种自然现象渲染特效。云特效渲染是指在三维模型表面上展示出具有云彩的图案。云彩效果是蓝色的天空上有一些白色的云，但如果用纯粹的蓝色得到的结果是没有云彩的感觉的，因为真实的云彩在蓝色的背景上还有白色的云。并且这些云的出现是不规律的，是随机的。在三维模型上就需要得到一些随机的数据来确定白色云彩出现的位置。并且白色云彩和蓝色背景之间的颜色是渐变的，通过生成的噪声数据就可以计算相应的像素颜色。云彩色 GLSL 代码如下所示。

1. 顶点着色器

```
varying float LightIntensity;
varying vec3 MCposition;
uniform vec3 LightPos;
uniform float Scale;

void main(void)
{
    vec4 ECposition  = gl_ModelViewMatrix * gl_Vertex;
    MCposition       = vec3(gl_Vertex) * Scale;
    vec3 tnorm       = normalize(vec3(gl_NormalMatrix * gl_Normal));
    LightIntensity   = dot(normalize(LightPos - vec3(ECposition)),tnorm) * 1.5;
    gl_Position      = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

2. 像素着色器

```
varying float LightIntensity;
varying vec3 MCposition;

uniform sampler3D Noise;
uniform vec3 Offset;
uniform vec3 SkyColor;
uniform vec3 CloudColor;

void main(void)
{
    vec4  noisevec    = texture3D(Noise, MCposition + Offset);
    float intensity   = (noisevec[0] + noisevec[1] +
                         noisevec[2] + noisevec[3]) * 1.5;
    vec3 color    = mix(SkyColor, CloudColor, intensity) * LightIntensity;
    color    = clamp(color, 0.0, 1.0);
    gl_FragColor = vec4(color, 1.0);
}
```

3. 噪声生成和参数传递

参数传递和其他特效方法一样，但这里需要在生成相应的 GPU 程序时生成柏林噪声贴图，然后通过参数传递的方法传递给 GLSL 程序，从而可以在之后的 GLSL 程序中进行调用。代码如下。

```
public GPURenderCloudOne()
    {
        VSFileName = "GLSL/Noise/Cloud1.vert";
        FSFileName = "GLSL/Noise/Cloud1.frag";

        PerlinNoise noise = new PerlinNoise();
        noise.CreateNoise3D();
    }

    public override void PassParameter()
    {
        if(ConfigGPUCommon.Instance.EnableTime)
        {
            ConfigGPUCloud.Instance.Scale =
                (ConfigGPUCloud.Instance.Scale + 0.01f) % 1;
        }
        GL.Uniform3(GL.GetUniformLocation(this.handle, "LightPos"),
            ConfigGPUCommon.Instance.LightPosition);
```

```
            GL. Uniform3( GL. GetUniformLocation( this. handle, "SkyColor" ),
                ConfigGPUCloud. Instance. SkyColor );
            GL. Uniform3( GL. GetUniformLocation( this. handle, "CloudColor" ),
                ConfigGPUCloud. Instance. CloudColor );
            GL. Uniform1( GL. GetUniformLocation( this. handle, "Scale" ),
                ConfigGPUCloud. Instance. Scale);
            GL. Uniform1( GL. GetUniformLocation( this. handle, "Noise" ),0);
        }
    }
```

4. 效果图

如图 7-8 所示，第一行是同一个三维模型不同参数得到的云特效，第二行是不同模型的云特效渲染。从中可以看出，通过噪声，可以在原来比较单一的颜色上生成没有规律的起伏，从而可以模拟云特效的感觉。

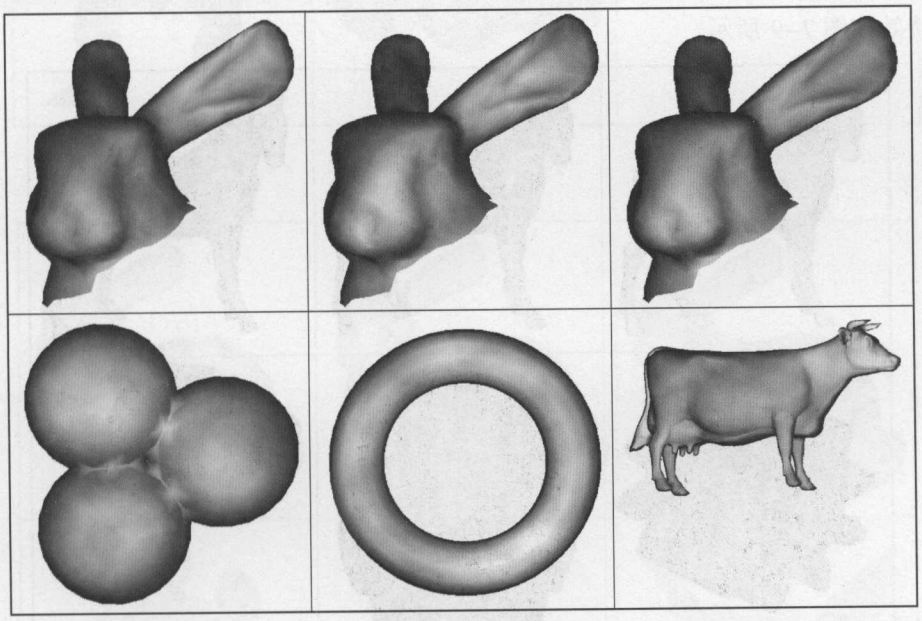


图 7-8　云特效渲染

5. 火的模拟

火的模拟和云彩的模拟计算方法类似，也需要用到噪声，从而可以得到不规律的火焰。只是使用噪声的方法不一样。具体的计算代码如下。

```
varying float LightIntensity;
varying vec3 MCposition;
uniform sampler3D Noise;
uniform vec3 Color1;
uniform vec3 Color2;
uniform float NoiseScale;
```

```
uniform vec3 Offset;

void main(void)
{
    vec4 noisevec = texture3D(Noise,NoiseScale * (MCposition + Offset));
    float intensity = abs(noisevec.x - 0.25) +
                      abs(noisevec.y - 0.125) +
                      abs(noisevec.z - 0.0625) +
                      abs(noisevec.w - 0.03125);
    intensity      = clamp(intensity * 6.0,0.0,1.0);
    vec3 color     = mix(Color1,Color2,intensity) * LightIntensity;
    color = clamp(color,0.0,1.0);
    gl_FragColor  = vec4(color,1.0);
}
```

火特效如图 7-9 所示。

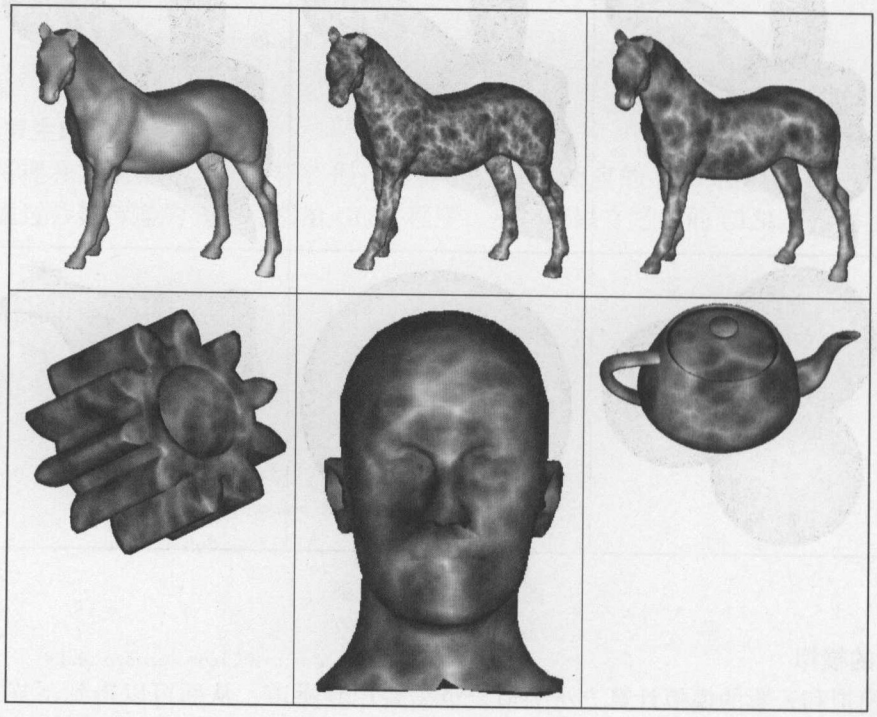


图 7-9　火特效

6. 木头纹理特效

除了自然现象之外，柏林噪声还可以用于其他各种各样的特效渲染。如图 7-10 所示，第一行是同一个三维模型在不同参数下的木头纹理效果；第二行是不同三维模型渲染得到的木头纹理效果。

7. 大理石特效

如图 7-11 所示，第一行是同一个三维模型在不同参数下大理石纹理效果；第二行是不

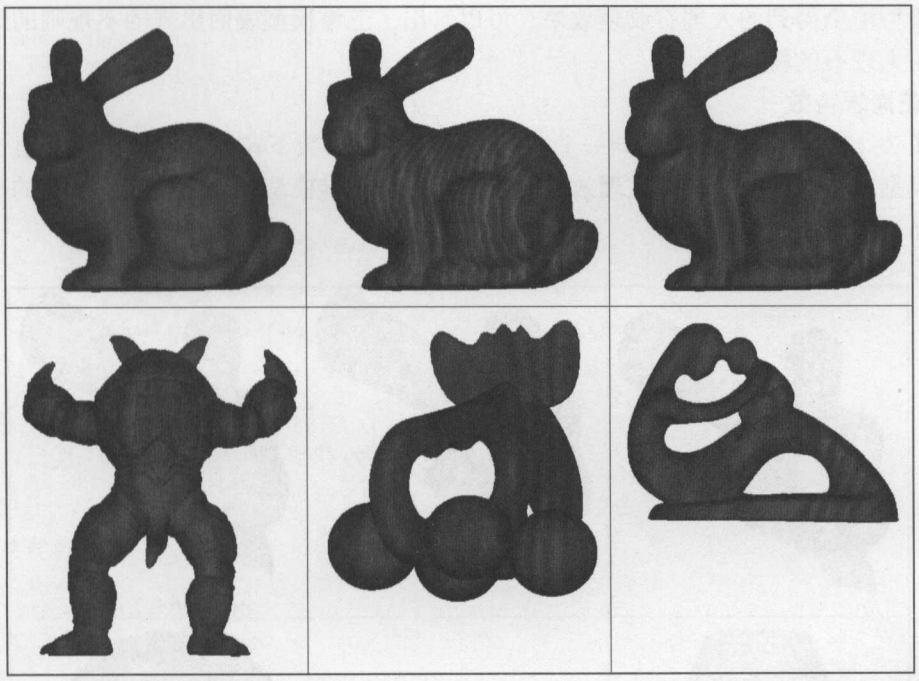


图 7-10　木头特效

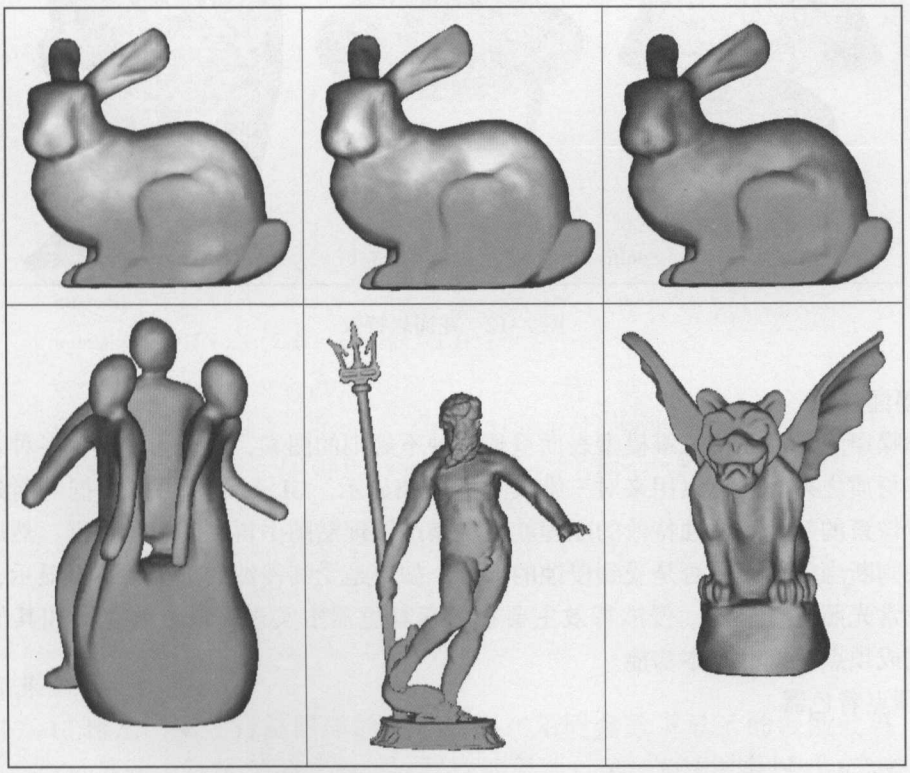


图 7-11　大理石特效

同三维模型渲染得到的大理石纹理效果。可以看出，三维模型表面出现的不规则的斑点更好地显示出大理石的特点。

8. 花岗岩特效

如图 7-12 所示，第一行是同一个三维模型在不同参数下花岗岩纹理效果；第二行是不同三维模型渲染得到的花岗岩纹理效果。可以看出，三维模型表面出现的不规则的凸起更好地显示出花岗岩的质地。

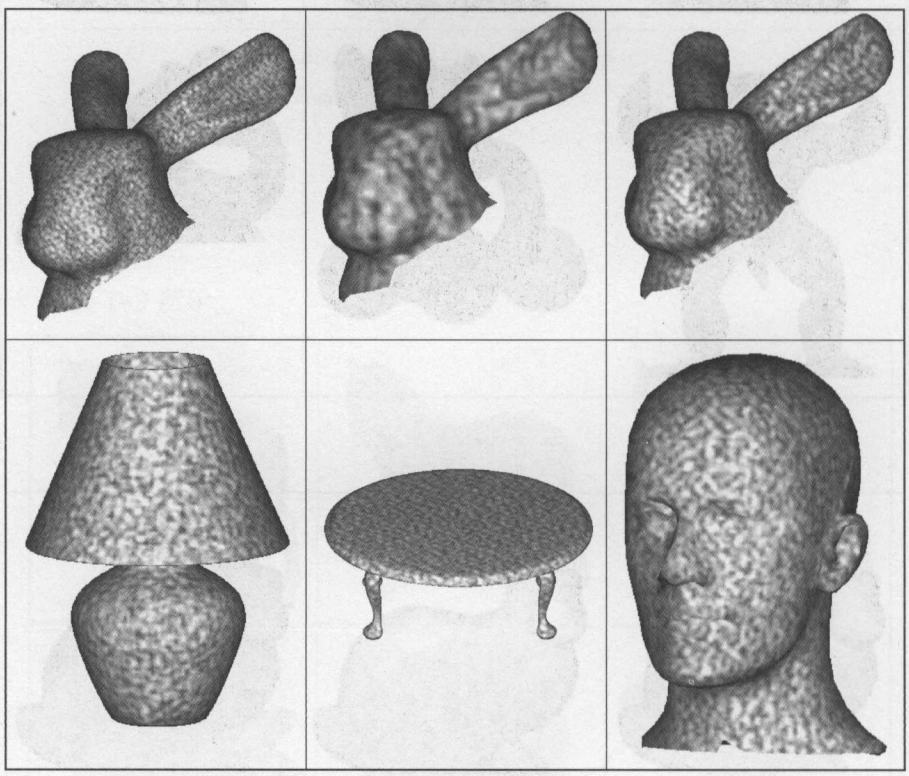


图 7-12　花岗岩特效

9. 侵蚀特效

柏林噪声除了可以使三维模型表面呈现各种不规则的图案，从而可以模拟各种自然界存在的真实材质之外，还可以用来对三维模型做侵蚀显示。GLSL 语言可以根据判断条件来显示和设置像素的颜色。侵蚀特效的原理就是从噪声纹理贴图中得到随机的数据，然后根据这些数据来判断当前像素是否是受到侵蚀的部分。如果是受到侵蚀的部分就不再显示，其他部分按照正常光照进行显示。侵蚀特效主要在像素着色器中实现，顶点着色器和其他特效类似，只完成顶点变换等基本功能。

1）顶点着色器

```
varying float lightIntensity;
varying vec3 Position;
uniform vec3 LightPosition;
```

```
uniform float Scale;

void main(void)
{
    vec4 pos = gl_ModelViewMatrix * gl_Vertex;
    Position = vec3(gl_Vertex) * Scale;
    vec3 tnorm = normalize(gl_NormalMatrix * gl_Normal);
    float dotval = max(dot(normalize(LightPosition - vec3(pos)),
                           tnorm),0.0);
    lightIntensity = dotval * 1.5;
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

2）像素着色器

```
varying float lightIntensity;
varying vec3 Position;
uniform vec3 Offset;
uniform sampler3D sampler3d;

void main(void)
{
    vec4 noisevec;
    vec3 color;
    float intensity;
    noisevec = texture3D(sampler3d,1.2 * (vec3(0.5) + Position));
    intensity = 0.75 * (noisevec.x + noisevec.y + noisevec.z + noisevec.w);
    intensity = 1.95 * abs(2.0 * intensity - 1.0);
    intensity = clamp(intensity,0.0,1.0);
    if(intensity < fract(0.5 - Offset.x - Offset.y - Offset.z)) discard;
    color = mix(vec3(0.2,0.1,0.0),vec3(0.8,0.8,0.8),intensity);
    color *= lightIntensity;
    color = clamp(color,0.0,1.0);
    gl_FragColor = vec4(color,1.0);
}
```

3）效果图

如图7-13所示，第一行是同样的三维模型在不同参数下显示的侵蚀效果，可以看出，随着参数的变化，三维模型出现不同程度的侵蚀；第二行是不同三维模型的侵蚀渲染效果。

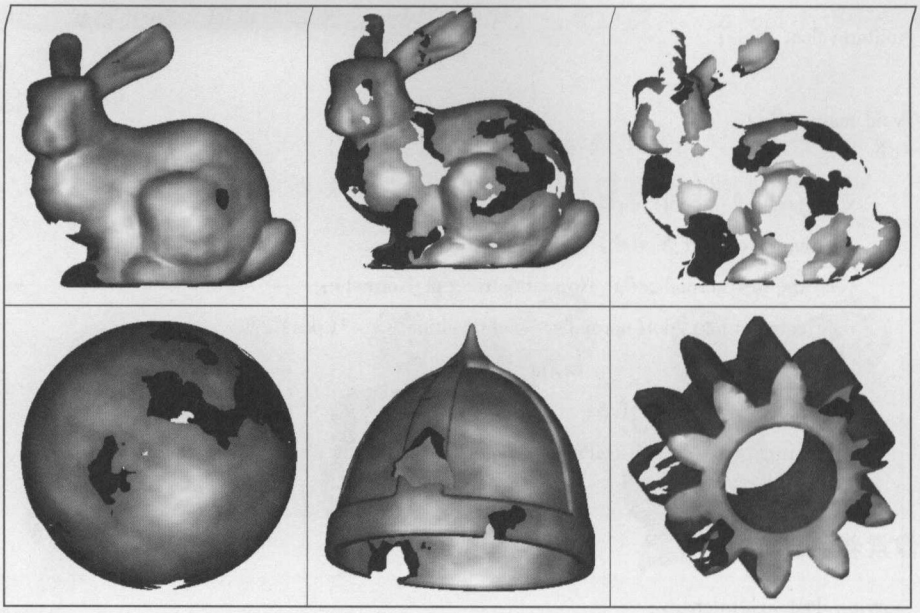


图 7-13　侵蚀特效

第8章 基于过程渲染

8.1 条纹渲染

第8章的渲染效果的基础是没有规律的噪声数据，这样可以显示出一些真实的材质。还有很多渲染效果基于有规律的模板，通过这些模板可以使三维模型表面呈现特定的、有规则的图案。基于过程的渲染使用代码来生成三维模型表面显示的纹理，而基于贴图的渲染所使用的纹理是预先存储在图像中的。基于过程的渲染需要更少的内存，并且代码动态生成的纹理不存在分辨率的问题，可以生成任意精度的纹理。而假如使用预先保存在图像中的纹理，显示的分辨率就由图像所决定，并且代码动态生成的纹理没有缝隙或重复等不自然的缺陷。还可以通过参数来控制代码生成纹理的结果，改变参数，生成的纹理就不一样。而图像纹理无法实现这样灵活多变的功能。基于过程渲染的缺点是必须进行编程才能实现特定的纹理，每种纹理需要不同的编程，这样就需要更多的编码技术。由于使用的函数可能基于不同的平台得到的结果不一样，因此基于过程渲染得到的纹理可能在不同的平台下结果不一样。基于过程的渲染常常用在墙面、地板、土地等纹理渲染上，而不能用在绘画、文字等艺术效果的纹理渲染上。也可以采用混合的办法，把基于图像的纹理和基于过程的纹理结合起来，也就是背景纹理用基于过程的方法生成，然后前景用基于图像的纹理进行合成。

条纹渲染是一种常见的基于过程的渲染技术。很多人造物体可以用条纹渲染来显示，如墙纸、旗帜、儿童玩具、编织物等。条纹渲染的原理是采用纹理坐标的 s 坐标或 t 坐标来显示条纹。根据纹理坐标数值的大小，来判断是否显示当前像素为条纹颜色，还是背景颜色。如果三维模型没有纹理坐标，那么可以使用三维模型的顶点坐标作为纹理坐标来进行判断。由于顶点坐标是严格具有规律的，因此生成的条纹大小和间隔都是一样的，而纹理坐标生成的条纹大小和间隔受到纹理坐标分布的影响。其中顶点着色器用来计算光照和把纹理坐标传递给像素着色器。生成条纹的算法都在像素着色器中实现。

1. 算法步骤

1）顶点着色器

顶点着色器负责光照的计算，包括漫反射和镜面反射颜色值。

第一步：计算顶点眼睛坐标和法向量。

```
vec3 ecPosition    = vec3( gl_ModelViewMatrix * gl_Vertex );
vec3 tnorm         = normalize( gl_NormalMatrix * gl_Normal );
```

第二步：根据眼睛坐标计算光线矢量和查看方向矢量。

```
vec3 lightVec    = normalize(LightPosition - ecPosition);
vec3 viewVec     = normalize(EyePosition - ecPosition);
```

第三步：计算光线矢量和查看方向矢量的矢量和。

```
vec3 Hvec    = normalize(viewVec + lightVec);
```

第四步：利用光线矢量和查看方向矢量的矢量和与法向量求镜面反射强度，并限制在(0,1) 之间，然后进行锐化。

```
float spec       = clamp(dot(Hvec,tnorm),0.0,1.0);
Spec             = pow(spec,16.0);
```

第五步：计算漫反射和镜面反射颜色值。

```
DiffuseColor      = LightColor * vec3(Kd * dot(lightVec,tnorm));
DiffuseColor      = clamp(Ambient + DiffuseColor,0.0,1.0);
SpecularColor     = clamp((LightColor * Specular * spec),0.0,1.0);
```

第六步：计算纹理坐标。像素着色器中绘制条纹颜色或背景颜色就是根据纹理坐标绘制的。

```
if(gl_MultiTexCoord0.s == 0&&gl_MultiTexCoord0.t == 0)
  gl_TexCoord[0] = gl_Vertex;
else
  gl_TexCoord[0] = gl_MultiTexCoord0;
```

第七步：这是每一个顶点着色器都必须做的事情，计算齐次顶点位置。

```
gl_Position      = ftransform();
```

2）像素着色器

像素着色器负责计算条纹绘制的位置及最终的着色。

第一步：将传入的 t 纹理坐标与条纹大小因子相乘并得到小数部分，表示条纹图案绘制的位置。Scale 值越大计算结果会得到越多的条纹。

```
float scaled_t      = fract(gl_TexCoord[0].t * Scale);
```

第二步：计算条纹颜色和背景颜色之间过渡的位置，避免走样。条纹两侧都要计算条纹边界位置。

```
float frac1      = clamp(scaled_t/Fuzz,0.0,1.0);
float frac2      = clamp((scaled_t - Width)/Fuzz,0.0,1.0);
```

第三步：根据第二步求出的条纹边界位置，执行一次线性混合，并在线性混合的基础上做一次埃尔米插值，这样条纹颜色之间的过渡区域产生一个平滑模糊的边界。

```
frac1    = frac1 * (1.0 - frac2);
frac1    = frac1 * frac1 * (3.0 - (2.0 * frac1));
```

第四步：应用顶点着色器提供的漫反射和镜面反射计算最终的着色。

```
vec3 finalColor = mix(BackColor,StripeColor,frac2);
finalColor      = finalColor * DiffuseColor + SpecularColor;
```

第五步：计算内置变量 gl_FragtColor 值，即最后输出的颜色。

```
gl_FragColor = vec4(finalColor,1.0);
```

2. 参数界面

如图 8-1 所示是条纹渲染中要用到的参数。通过改变这些参数就可以生成不同的条纹渲染效果。条纹渲染中和生成条纹相关的参数有条纹颜色、背景颜色、条纹宽度、缩放倍数、模糊系数。其中缩放倍数决定显示的条纹数量，条纹宽度是条纹和背景的对比系数，例如，0.5 表示条纹宽度和背景宽度一样。模糊系数控制条纹颜色和背景颜色渐变的过程。

Ambient	(0.2, 0.2, 0.2)
BackColor	(0.2, 0.2, 1)
EyePosition	(0, 0, 4)
Fuzz	0.1
Kd	0.8
LightColor	(0.9, 0.8, 0.7)
Scale	10
Specular	(0.2, 0.2, 0.2)
StripeColor	(1, 0.5, 0)
Width	0.5

图 8-1　条纹渲染参数界面

3. 完整代码

1）顶点着色器

```
uniform vec3  LightPosition;
uniform vec3  LightColor;
uniform vec3  EyePosition;
uniform vec3  Specular;
uniform vec3  Ambient;
uniform float Kd;

varying vec3  DiffuseColor;
varying vec3  SpecularColor;

void main()
{
    vec3 ecPosition = vec3(gl_ModelViewMatrix * gl_Vertex);
    vec3 tnorm      = normalize(gl_NormalMatrix * gl_Normal);
    vec3 lightVec   = normalize(LightPosition - ecPosition);
    vec3 viewVec    = normalize(EyePosition - ecPosition);
    vec3 Hvec       = normalize(viewVec + lightVec);

    float spec = clamp(dot(Hvec,tnorm),0.0,1.0);
    spec = pow(spec,16.0);

    DiffuseColor    = LightColor * vec3(Kd * dot(lightVec,tnorm));
    DiffuseColor    = clamp(Ambient + DiffuseColor,0.0,1.0);
    SpecularColor   = clamp((LightColor * Specular * spec),0.0,1.0);
```

```
    if( gl_MultiTexCoord0. s == 0&&gl_MultiTexCoord0. t == 0)
        gl_TexCoord[0] = gl_Vertex;
    else
            gl_TexCoord[0]    = gl_MultiTexCoord0;
        gl_Position       = ftransform( );
    }
```

2）像素着色器

```
    uniform vec3   StripeColor;
    uniform vec3   BackColor;
    uniform float  Width;
    uniform float  Fuzz;
    uniform float  Scale;

    varying vec3   DiffuseColor;
    varying vec3   SpecularColor;

    void main( )
    {
        float scaled_t = fract( gl_TexCoord[0]. t * Scale) ;

        float frac1 = clamp( scaled_t/Fuzz,0. 0,1. 0) ;
        float frac2 = clamp( ( scaled_t - Width)/Fuzz,0. 0,1. 0) ;

        frac1 = frac1 * ( 1. 0 - frac2) ;
        frac1 = frac1 * frac1 * ( 3. 0 - ( 2. 0 * frac1) ) ;

        vec3 finalColor = mix( BackColor,StripeColor,frac2) ;
        finalColor = finalColor * DiffuseColor + SpecularColor;

        gl_FragColor = vec4( finalColor,1. 0) ;
    }
```

4. 效果图

1）实验一

如图 8-2 所示，第一行是同一个三维模型在不同参数下的条纹渲染结果，从中可以看出，通过改变参数，同样的代码可以显示出不同的渲染效果；第二行是不同的三维模型的条纹渲染结果。图 8-2 所示的三维模型都没有纹理坐标，因此采用顶点坐标作为条纹的生成条件，从而得到的条纹大小和间隔都很平均。

2）实验二

图 8-3 所示是具有纹理坐标的三维模型条纹渲染结果，从中可以看出，条纹大小、间隔的出现受到纹理坐标的影响。

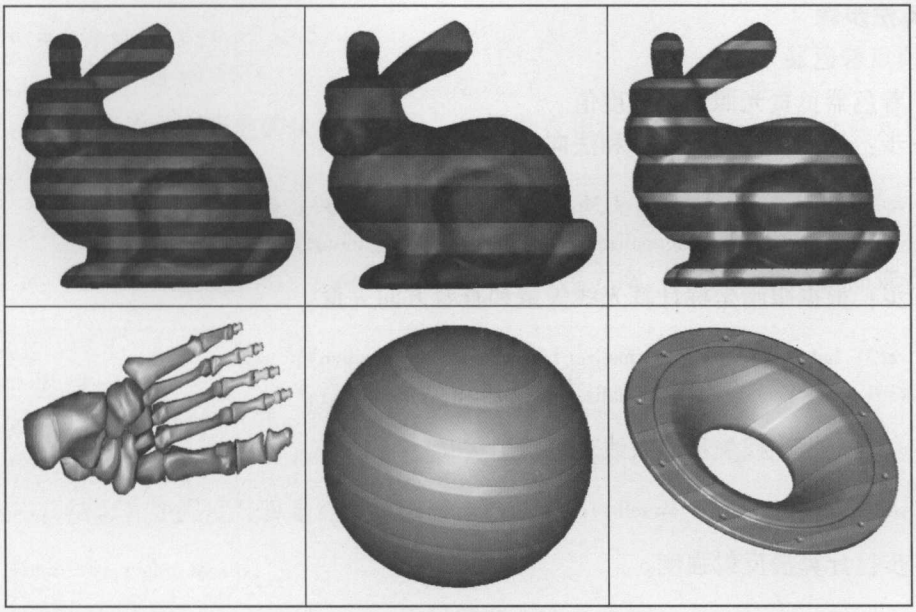


图 8-2　条纹渲染效果

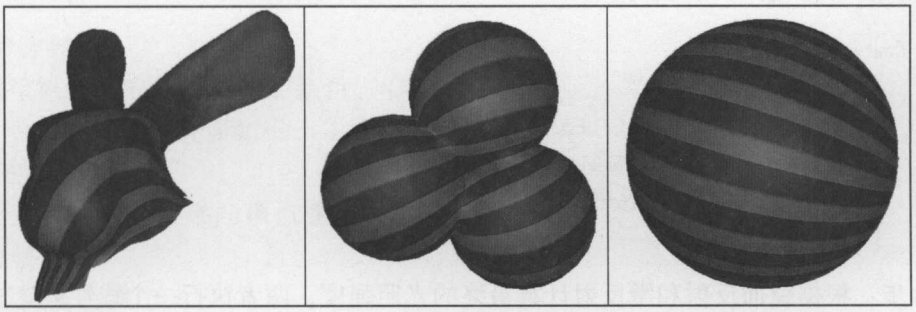


图 8-3　具有纹理坐标的条纹渲染效果

8.2　砖墙渲染效果

砖墙渲染是一个会对所有模型对象都应用一个计算后的砖墙图案的着色器。砖墙渲染不是一个看起来最真实的砖块着色器，而是一个相当简单的着色器，程序不会对这个砖块图案使用纹理，图案本身将通过算法生成。通过将砖块算法的不同方面参数化，可以使这个着色器具有一定的灵活性。砖墙的渲染效果和条纹类似，也是一种有规律的模式，但和条纹不一样的地方是，砖墙需要两个方向来计算砖的大小和纹理，而条纹只需要一个方向。同时为了使砖的大小和分布一致，因此砖墙采用三维模型的顶点 x、y 坐标来进行计算，如果采用纹理坐标就会造成砖的大小不一。交替的砖块行将会错开半个砖块的宽度，还会计算漫反射和镜面反射特征，并且要有控制砖块颜色、灰泥颜色、砖块与砖块的水平距离、砖块与砖块的垂直距离、砖块宽度分数（砖块的宽度相对于两个相邻砖块之间的总体水平距离的比例）及砖块高度分数（砖块的高度相对于两个相邻砖块之间的总体垂直距离的比例）的参数。

1. 算法步骤

1）顶点着色器

顶点着色器负责光照计算强度值。

第一步：计算顶点眼睛坐标和法向量。

```
vec3 ecPosition     = vec3(gl_ModelViewMatrix * gl_Vertex);
vec3 tnorm          = normalize(gl_NormalMatrix * gl_Normal);
```

第二步：根据眼睛坐标计算光线矢量和查看方向矢量。

```
vec3  lightVec      = normalize(LightPosition - ecPosition);
vec3  viewVec       = normalize(EyePosition - ecPosition);
```

第三步：根据光线矢量和法线计算反射矢量。

```
vec3  reflectVec    = reflect( -lightVec,tnorm);
```

第四步：计算漫反射强度。

```
float     diffuse   = max(dot(lightVec,tnorm),0.0);
```

第五步：计算镜面反射强度。

```
float spec          =0.0;
  if(diffuse >0.0)
    {
        spec = max(dot(reflectVec,viewVec),0.0);
        spec = pow(spec,16.0);
    }
```

第六步：根据镜面反射和漫反射计算最终的光照强度，两者执行一个线性运算。

```
LightIntensity      = intensity(diffuse,spec);
```

第七步：计算顶点的坐标的位置，用于像素着色器绘制砖块。

```
MCposition      = gl_Vertex.xy;
```

第八步：计算齐次顶点位置。

```
gl_Position     = ftransform();
```

2）像素着色器

像素着色器负责计算砖块图案。

第一步：计算绘制砖块的位置，包括行、列位置，即砖块行号和砖块所在的那一行中的砖块号。

```
vec2   position;
position = MCposition/BrickSize;
```

第二步：判断片元是否偏移了一行，如果偏移则改变砖块号，偏移半块砖。

```
if( fract( position. y * 0. 5) >0. 5)
    position. x +=0. 5;
```

第三步：计算片元在当前砖块中的位置，包括垂直和水平位置。这将用作确定使用砖块颜色还是灰泥颜色的基础。

```
position = fract( position) ;
```

第四步：用 step 函数返回 0、1 两种值实现两种着色方案，用 mix 函数返回选择了哪一种着色。

```
vec2   userBrick;
useBrick  = step( position, BrickPct) ;
color     = mix( MortarColor, BrickColor, useBrick. x * useBrick. y) ;
```

第五步：结合光照强度计算最终的着色值。

```
Color   *= LightIntensity;
```

第六步：计算内置变量 gl_FragtColor 值，即最后输出的颜色。

```
gl_FragColor     = vec4( color, 1. 0) ;
```

2. 参数界面

砖墙渲染用到的参数有砖的颜色、墙的颜色、砖的大小、砖的间隔。界面如图 8-4 所示。

BrickColor	(1, 0.3, 0.2)
BrickPct	(0.8, 0.75)
BrickSize	(0.3, 0.15)
MortarColor	(0.85, 0.86, 0.84)

图 8-4　砖墙效果参数界面

3. 完整代码

下面是砖墙渲染效果的顶点着色器和像素着色器的完整代码。

1）顶点着色器

```
uniform vec3 LightPosition;
const float SpecularContribution  =0. 3;
const float DiffuseContribution   =1. 0 - SpecularContribution;
varying float LightIntensity;
varying vec2 MCposition;
void main( void)
{
    vec3 ecPosition  = vec3( gl_ModelViewMatrix * gl_Vertex) ;
    vec3 tnorm       = normalize( gl_NormalMatrix * gl_Normal) ;
    vec3 lightVec    = normalize( LightPosition - ecPosition) ;
    vec3 reflectVec  = reflect( -lightVec, tnorm) ;
    vec3 viewVec     = normalize( -ecPosition) ;
    float diffuse    = max( dot( lightVec, tnorm) , 0. 0) ;
    float spec       =0. 0;
    if( diffuse >0. 0)
```

```
    {
        spec = max( dot( reflectVec, viewVec ), 0.0 );
        spec = pow( spec, 16.0 );
    }
    LightIntensity  = DiffuseContribution * diffuse +
                      SpecularContribution * spec;
    MCposition      = gl_Vertex.xy;
    gl_Position     = ftransform( );
}
```

2）像素着色器

```
uniform vec3  BrickColor, MortarColor;
uniform vec2  BrickSize;
uniform vec2  BrickPct;
varying vec2  MCposition;
varying float LightIntensity;

void main( void )
{
    vec3  color;
    vec2  position, useBrick;
    position = MCposition / BrickSize;
    if( fract( position.y * 0.5 ) > 0.5 )
        position.x += 0.5;
    position = fract( position );
    useBrick = step( position, BrickPct );
    color    = mix( MortarColor, BrickColor,
    useBrick.x * useBrick.y );
    color *= LightIntensity;
    gl_FragColor = vec4( color, 1.0 );
}
```

4. 效果图

如图 8-5 所示，第一行是兔子头三维模型在不同参数下的砖墙渲染效果；第二行是不同的三维模型砖墙渲染效果。

5. 反走样砖墙渲染

由于计算机以离散点生成图形，生成的图形必然与真实景物存在差距，这种差距表现为：直线或光滑曲面的锯齿、花纹失去原有色彩形状、细小物体在画面的消失等，这些全部称为走样（Aliasing）。反走样可以减少这种情况，就是把原来边界的地方锯齿部分用低饱和度的点补上，这样既不影响整体轮廓，又能获得较好的平滑效果。用于减少和消除各种走样现象的方法就是反走样。

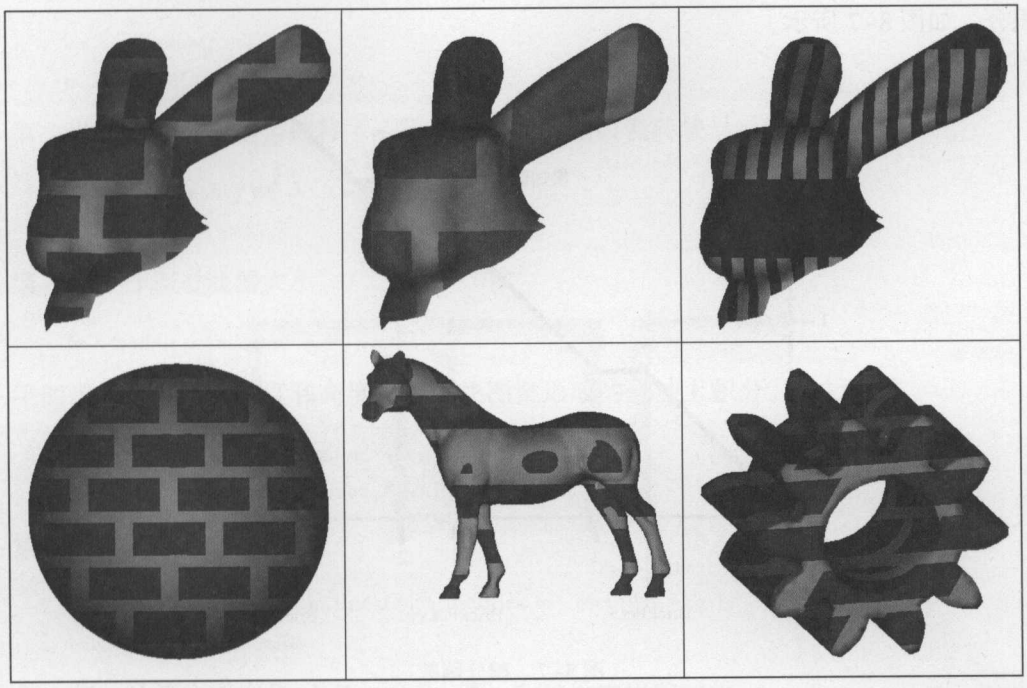


图 8-5　砖墙渲染效果

前面介绍的砖墙渲染从砖块颜色到灰泥颜色是突变的，当时用的是 step 函数返回 0、1 两种值代表两种着色，其函数就是一个周期性的步进函数或一个脉冲串，从 0 到 1 或从 1 到 0 都是骤减的。在水平方向上创建砖墙图案的函数如图 8-6 所示。

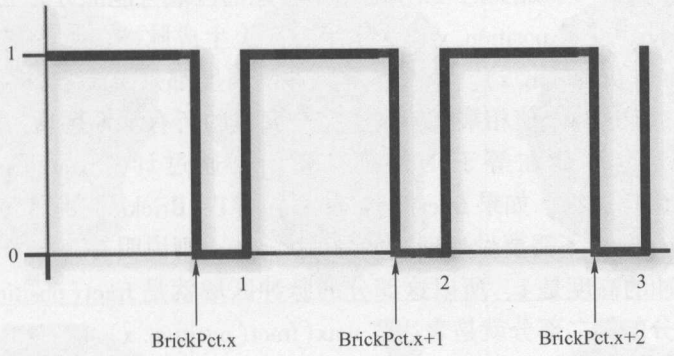


图 8-6　脉冲函数

现在利用积分的方法进行反走样。fwidth 函数是内置函数，其返回值是某点 x、y 坐标变化率绝对值的和，即

$$\text{fwidth}(v) = \text{abs}(\text{dFdx}(v)) + \text{abs}(\text{dFdy}(v))$$

其中 dFdx(v)和 dFdy(v)也是 GLSL 中的内置函数，分别提供了坐标在 x 方向上的变化速度和在 y 方向上的变化速度。过滤器可以理解为之前用到的 step 函数或 smoothstep 函数，用于做平滑处理，避免走样。fwidth 函数计算的过滤器宽度将覆盖几个脉冲。通过积分而不是对函数的采样，将会得到一个适当的加权平均值，并避免由点采样导致的走样效果。对函数进

行积分，如图 8-7 所示。

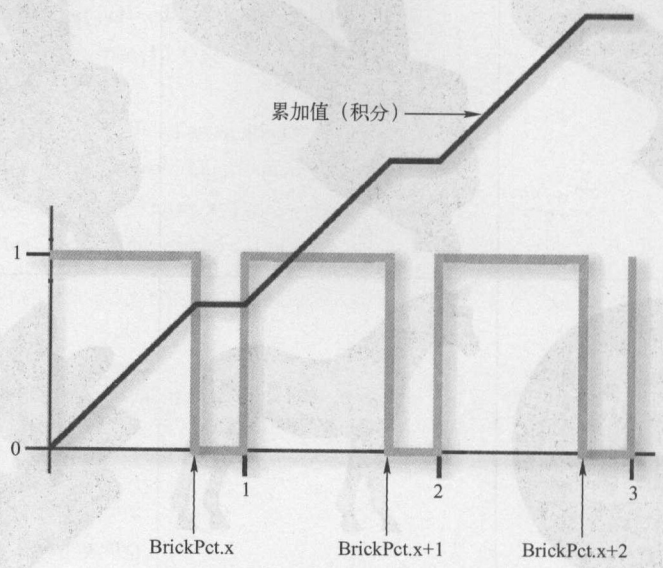


图 8-7　积分函数

从 0 到 BrickPct. x，函数值为 1，所以积分增加，斜率为 1；从 BrickPct. x 到 1，函数值为 0，所以积分在这个区域保持不变，可以看出其函数积分是一种倾斜和平稳的叠加模式。

通过确定积分在过滤器的区域上的值，就可以执行反走样了。这是通过计算在过滤器的边缘上的积分并减去两个值来实现的。这个函数的积分由两部分组成：在所考虑的边缘之前已经全部完成的所有脉冲区域的和，以及对于所考虑的区域可能部分完成的脉冲区域。在砖墙着色器中，可以使用变量 position. x 作为水平方向上生成脉冲函数的基础，所以完全完成的脉冲区域就是 floor（position. x）。因为各个脉冲的高度都是 1，所以各个完全完成的脉冲区域是 BrickPct. x。将这两个值相乘就得到了完全完成的所有脉冲区域。边缘可能在函数等于 0 的那一部分，也可能在等于 1 的那一部分。通过计算 fract（position. x）-（1 - BrickPct. x）就可以确定边缘。如果 fract（position. x）-（1 - BrickPct. x）计算的值小于 0，则处在函数中返回 0 的部分，不需要做处理。如果值大于 0，则说明有一部分已经进入函数等于 1 的区域。因为脉冲的高度是 1，所以这部分的脉冲区域就是 fract（position. x）-（1 - BrickPct. x）。因此，积分的第二部分就是表达式 max（fract（position. x）-（1 - BrickPct. x），0）。

接下来就可以对砖墙图案的水平和垂直部分使用这个积分了。因为应用程序知道砖块的宽度和高度部分，所以很容易计算 1 - BrickPct. x 和 1 - BrickPct. y，并将它们提供给像素着色器。

积分的累加可以定义为一个宏或函数：#define Integral（x，p，notp）（floor（x）*（p）+ max（fract（x）-（notp））），参数 p 是属于脉冲的值，而 notp 是不属于脉冲的值。使用这个宏可以编写代码来计算在过滤器宽度上的积分值。

1）反走样像素着色器

第一步：计算绘制砖块的位置，包括行、列位置，即砖块行号和砖块所在那一行中的砖块号。

```
vec2    position;
position = MCposition/BrickSize;
```

第二步：判断片元是否偏移了一行，如果偏移则改变砖块号，偏移半块砖。

```
if( fract( position. y * 0. 5) >0. 5)
    position. x +=0. 5;
```

第三步：计算过滤器大小。

```
fw = fwidth( position);
```

第四步：在过滤器宽度和高度上对砖块图案形成的脉冲上积分，执行过滤。

```
useBrick    = ( Integral( position + fw, BrickPct, MortarPct)    -
                Integral( position, BrickPct, MortarPct) )/fw;
```

第五步：确定最终的颜色。

```
color    = mix( MortarColor, BrickColor, useBrick. x * useBrick. y);
color    *= LightIntensity;
```

第六步：计算内置变量 gl_FragtColor 值，即最后输出的颜色。

```
gl_FragColor    = vec4( color, 1. 0);
```

2）反走样效果图

反走样砖墙渲染效果如图 8-8 所示。

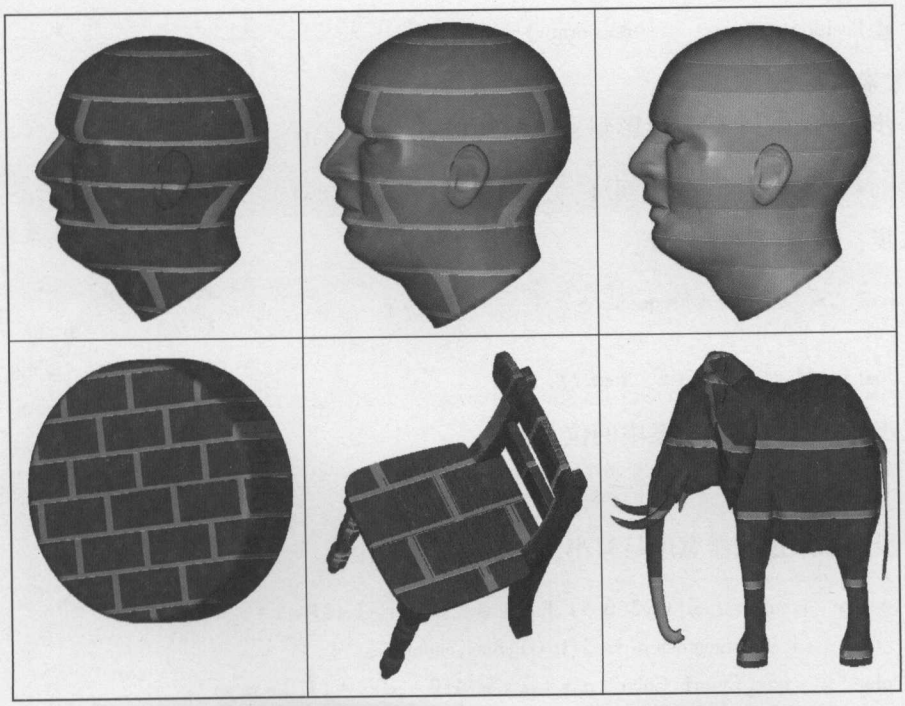


图 8-8　反走样砖墙渲染效果

8.3 棋盘渲染

棋盘渲染是指在三维模型表面生成黑白间隔的类似棋盘的图案。当然，除了黑白色，程序设置了外部变量，用户可以设置自己喜欢的颜色。棋盘渲染并不需要光照计算，所以顶点着色器只需传入纹理坐标即可。应用程序会提供棋盘图案的两种颜色值、这两种颜色的平均值（应用程序可以计算这个值并通过一个一致变量提供它，而不是让像素着色器对每一个片元都计算这个值）及棋盘图案的频率。为了避免棋盘两种颜色的生硬过渡，着色器会计算过滤器的适当大小并使用它在邻近的棋盘方块之间执行平滑的插值，如果过滤器太宽则会使用平均颜色，所以像素着色器会使用一个条件语句，在 if 子句将计算好的颜色和平均颜色之间执行平滑插值，在 else 子句只使用平均颜色。

1. 算法步骤

1）顶点着色器

顶点着色器只需传入纹理坐标即可，并不进行光照计算。

第一步：计算纹理坐标。像素着色器中绘制棋盘就是根据纹理坐标绘制的。

```
if(gl_MultiTexCoord0.s==0&&gl_MultiTexCoord0.t==0)
    TexCoord=gl_Vertex.xy;
else
    TexCoord      =gl_MultiTexCoord0.st;
```

第二步：这是每一个顶点着色器都必须做的事情，计算齐次顶点位置。

```
gl_Position                =ftransform();
```

2）像素着色器

第一步：确定一个像素投影到 s－t 空间的宽度。

```
vec2 fw    =fwidth(TexCoord);
```

第二步：确定模糊的数量。

```
vec2 fuzz       =fw*Frequency*2.0;

float fuzzMax=max(fuzz.s,fuzz.t);
```

第三步：确定在棋盘图案中的位置。

```
vec2 checkPos       =fract(TexCoord*Frequency);
```

第四步：如果过滤器宽度足够小，则计算图案颜色。

```
vec2 p    =smoothstep(vec2(0.5),fuzz+vec2(0.5),checkPos)+
          (1.0-smoothstep(vec2(0.0),fuzz,checkPos));
color     =mix(Color1,Color2,p.x*p.y+(1.0-p.x)*(1.0-p.y));
```

第五步：在接近边界时使平均颜色淡入。

```
color    = mix(color,AvgColor,smoothstep(0.125,0.5,fuzzMax));
```

第六步：否则只使用平均颜色。

```
color = AvgColor;
```

第七步：计算内置变量 gl_FragtColor 的值，即最后输出的颜色。

```
gl_FragColor      = vec4(color,1.0);
```

2. 参数界面

棋盘渲染特效中用到的参数有生成棋盘图案的两种颜色，用于反走样的过渡色及棋盘图案的频率。调整棋盘图案的频率可以得到不同密度大小的棋盘。界面如图 8-9 所示。

AvgColor	(0.5, 0.5, 0.5)
Color1	(1, 1, 1)
Color2	(0, 0, 0)
Frequency	16

图 8-9　棋盘渲染参数界面

3. 完整代码

1）顶点着色器

```
varying vec2 TexCoord;
void main()
{
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    if(gl_MultiTexCoord0.s==0&&gl_MultiTexCoord0.t==0)
        TexCoord = gl_Vertex.xy;
    else
        TexCoord      = gl_MultiTexCoord0.st;
}
```

2）像素着色器

```
uniform vec3   Color1;
uniform vec3   Color2;
uniform vec3   AvgColor;
uniform float  Frequency;
varying vec2   TexCoord;

void main()
{
    vec3 color;
    vec2 fw = fwidth(TexCoord);
    vec2 fuzz = fw * Frequency * 2.0;
    float fuzzMax = max(fuzz.s,fuzz.t);
```

```
        vec2 checkPos = fract( TexCoord * Frequency );

        if( fuzzMax < 0.5 )
        {
            vec2 p = smoothstep( vec2( 0.5 ), fuzz + vec2( 0.5 ), checkPos ) +
                     ( 1.0 - smoothstep( vec2( 0.0 ), fuzz, checkPos ) );
            color = mix( Color1, Color2, p.x * p.y + ( 1.0 - p.x ) * ( 1.0 - p.y ) );
            color = mix( color, AvgColor, smoothstep( 0.125, 0.5, fuzzMax ) );
        }
        else
        {
            color = AvgColor;
        }
        gl_FragColor = vec4( color, 1.0 );
    }
```

4. 效果图

如图 8-10 所示，第一行是兔子三维模型在不同参数下的棋盘渲染效果，由于参数的不同，显示的黑白格子数量、大小也不一样；第二行是不同三维模型的棋盘渲染效果。

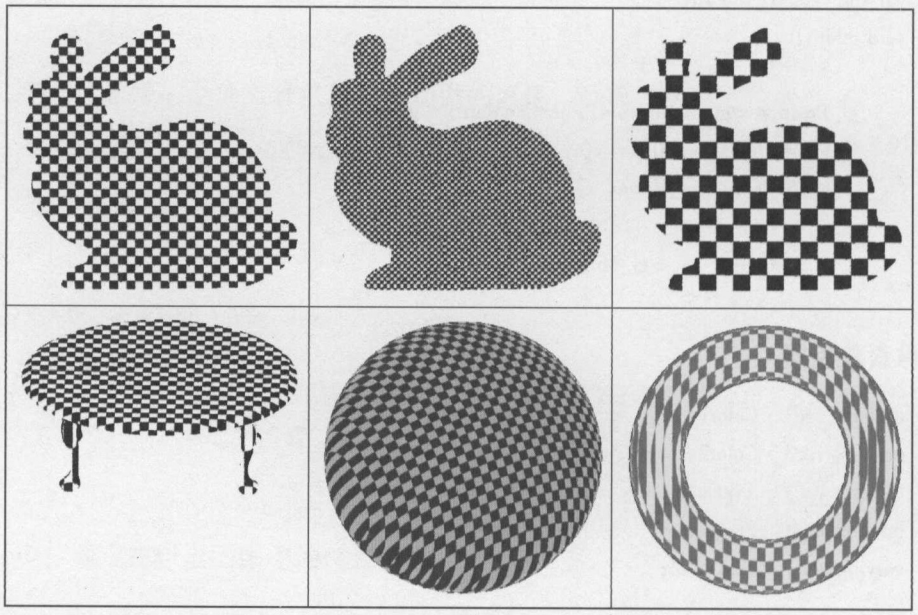


图 8-10　棋盘渲染效果

8.4　ToyBall 渲染

ToyBall 渲染是通过使用程序定义的星星图案和程序定义的条纹来对模型着色。这个着色器的关键在于星星图案是由定义了星星形状的 5 个半球空间的系数定义的。这些系数可使

星星图案的大小适合模型。相对于各个半球空间，模型上的点被分为“里”或者“外”。星星图案正中心的位置相对于全部5个半球空间来说都是“里”。星星的点中的位置相对于5个半球空间中的4个而言是“里”。条纹图案中片元的计算相对简单。在将表面上的各个位置分类为“星星”、“条纹”、“其他”之后，就可以相应地对各个片元着色了。这些颜色计算要按照一种顺序应用，以便确保即使是从很远的地方查看时也能产生一种合理的效果。表面法线是在像素着色器内部使用一种分析方法计算的。着色器利用了球体的性质（表面上任何点的表面法线所指的方向都与从球体中心到表面上的这个点的方向相同）来计算表面法线。每一个片元上还用了镜面高光。

1. 算法步骤

1）顶点着色器

顶点着色器只需提供顶点的位置，颜色和法线都是在像素着色器计算的。

第一步：计算顶点眼睛坐标。

```
ECposition          = gl_ModelViewMatrix * gl_Vertex;
```

第二步：眼睛坐标球体中心，为了让这个着色器正常工作，顶点着色器必须定义出一个球体。像素着色器将根据球体的已知几何形状执行计算，这个球体可以是任意大小的。

```
ECballCenter        = gl_ModelViewMatrix * BallCenter;
```

第三步：计算齐次顶点位置。

```
gl_Position         = ftransform( );
```

2）像素着色器

第一步：将所着色的表面位置转换为半径为1的球体上的一个点，P为着色器空间中的点。

```
p. xyz     = normalize( ECposition. xyz - ECballCenter. xyz) ;
p. w       = 1. 0;
```

第二步：初始化星星图案计数器。这里的初始值设为-3，因为定义了5个半球空间，所以计数器的范围为（-3,2）。值为1或2时表明片元在星星图案内部，否则在星星图案外部。

```
inorout    = InOrOutInit;
```

第三步：计算当前点与前4个半球空间的距离。

```
distance[0] = dot(p,HalfSpace0);
distance[1] = dot(p,HalfSpace1);
distance[2] = dot(p,HalfSpace2);
distance[3] = dot(p,HalfSpace3);
```

第四步：以计数器作为基础，实现星星图案颜色和球体表面其余部分颜色之间平滑反走样过渡。

```
distance        = smoothstep( - FWidth,FWidth,distance);
```

第五步：更新计数器的值，现在其值为（-3,1）。

```
inorout    += dot(distance,vec4(1.0));
```

第六步：计算到第 5 个半球空间的距离，确定片元是在球体周围的条纹“里”还是“外”。

```
distance.x    = dot(p,HalfSpace4);
distance.y    = StripeWidth - abs(p.z);
distance      = smoothstep( -FWidth,FWidth,distance);
```

第七步：更新计数器的值，现在其值为（-3,2）。

```
inorout       += distance.x;
```

第八步：将计数器的值限定为（0,1）。

```
inorout       = clamp(inorout,0.0,1.0);
```

第九步：计算片元的表面颜色。利用计数器执行黄色和红色的线性混合，并在此基础上与条纹颜色再执行一次线性混合。

```
surfColor     = mix(Yellow,Red,inorout);
surfColor     = mix(surfColor,Blue,distance.y);
```

第十步：计算法线，用于执行光照计算。利用了球体的性质：“表面上任何点的表面法线所指的方向都与从球体中心到表面上的这个点的方向相同。”

```
normal    = p;
```

第十一步：计算漫反射。

```
intensity       =0.2;
intensity      +=0.8 * clamp(dot(LightDir,normal),0.0,1.0);
surfColor      *= intensity;
```

第十二步：计算镜面反射。

```
intensity      = clamp(dot(HVector,normal),0.0,1.0);
intensity      = pow(intensity,SpecularColor.a);
```

第十三步：计算包含镜面反射和漫反射的最终着色值。

```
surfColor    += SpecularColor * intensity;
```

第十四步：计算内置变量 gl_FragtColor 的值，即最后输出的颜色。

```
gl_FragColor      = surfColor;
```

2. 参数界面

ToyBall 渲染特效中用到的参数有球体中心、反走样的过滤器宽度、条纹宽度、5 个半球空间、星星图案计数器。界面如图 8-11 所示。

BallCenter	(0, 0, 0, 1)
Blue	(0.5019608, 1, 0.5019608, 1)
FWidth	0.01
HalfSpace0	(1, 0.17, 0, 0.2)
HalfSpace1	(0.309017, 0.9510565, 0, 0.2)
HalfSpace2	(-0.809017, 0.5877852, 0, 0.2)
HalfSpace3	(-0.809017, -0.5877852, 0, 0.2)
HalfSpace4	(0.309017, -0.9510565, 0, 0.2)
HVector	(0.32506, 0.32506, 0.88808, 0)
InOrOutInit	-2.7
LightDir	(0.57735, 0.57735, 0.57735, 0)
Red	(0.7529412, 0, 0, 1)
SpecularColor	(0.4, 0.4, 0.4, 0.6)
StripeWidth	0.5
Yellow	(0, 0, 1, 1)

图 8-11　ToyBall 参数界面

3. 完整代码

下面是 ToyBall 渲染特效的顶点着色器和像素着色器完整代码。

1）顶点着色器

```
varying vec4 ECposition;
varying vec4 ECballCenter;
uniform vec4 BallCenter;

void main()
{
    ECposition   = gl_ModelViewMatrix * gl_Vertex;
    ECballCenter = gl_ModelViewMatrix * BallCenter;
    gl_Position  = ftransform();
}
```

2）像素着色器

```
varying vec4 ECposition;
varying vec4 ECballCenter;
uniform vec4  LightDir;
uniform vec4  HVector;
uniform vec4  SpecularColor;
uniform vec4  Red,Yellow,Blue;
uniform vec4  HalfSpace0;
uniform vec4  HalfSpace1;
uniform vec4  HalfSpace2;
uniform vec4  HalfSpace3;
uniform vec4  HalfSpace4;
uniform float InOrOutInit;
uniform float StripeWidth;
```

```
uniform float FWidth;
void main( )
{
    vec4   normal;
    vec4   p;
    vec4   surfColor;
    float intensity;
    vec4   distance;
    float inorout;
    p. xyz = normalize( ECposition. xyz - ECballCenter. xyz) ;
    p. w    =1. 0;
    inorout = InOrOutInit;
    distance[0] = dot( p, HalfSpace0) ;
    distance[1] = dot( p, HalfSpace1) ;
    distance[2] = dot( p, HalfSpace2) ;
    distance[3] = dot( p, HalfSpace3) ;
    distance = smoothstep( - FWidth, FWidth, distance) ;
    inorout + = dot( distance, vec4(1. 0)) ;
    distance. x = dot( p, HalfSpace4) ;
    distance. y = StripeWidth - abs( p. z) ;
    distance = smoothstep( - FWidth, FWidth, distance) ;
    inorout + = distance. x;
    inorout = clamp( inorout, 0. 0, 1. 0) ;
    surfColor = mix( Yellow, Red, inorout) ;
    surfColor = mix( surfColor, Blue, distance. y) ;
    normal = p;
    intensity    =0. 2; // ambient
    intensity + =0. 8 * clamp( dot( LightDir, normal) , 0. 0, 1. 0) ;
    surfColor  * = intensity;
    intensity    = clamp( dot( HVector, normal) , 0. 0, 1. 0) ;
    intensity    = pow( intensity, SpecularColor. a) ;
    surfColor + = SpecularColor * intensity;
    gl_FragColor = surfColor;
}
```

4. 效果图

如图 8-12 所示，第一行是兔子三维模型在不同参数下的渲染效果；第二行是不同的三维模型 ToyBall 渲染效果。其中的五角星图案随着视角的变换一直显示在三维模型的正面。

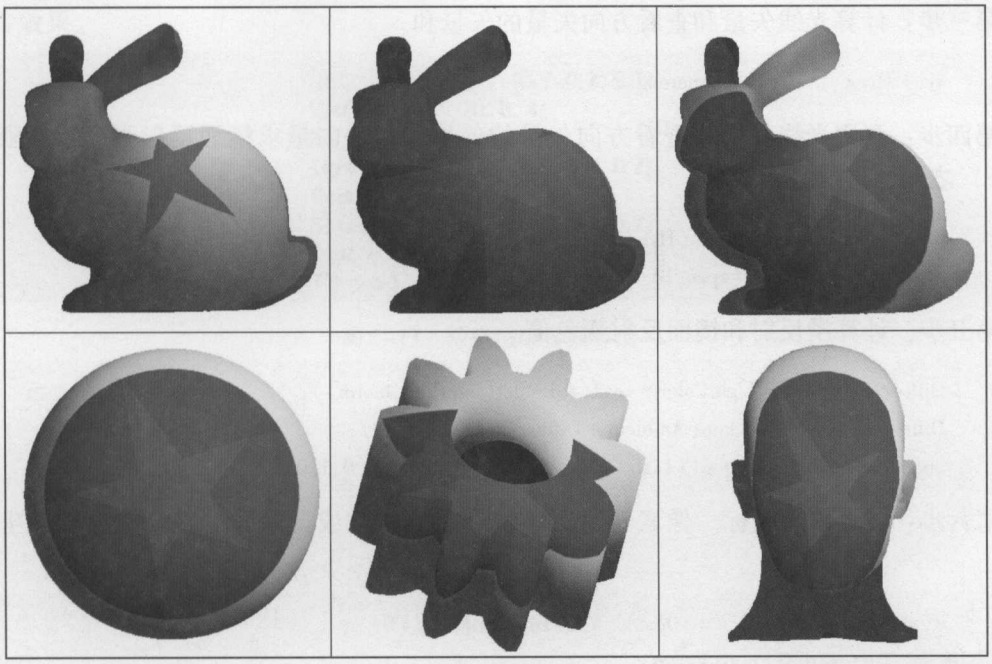


图 8-12　ToyBall 渲染效果

8.5　网格渲染

三维模型的渲染除了可以在三维模型表面显示图案之外，还可以直接更改三维模型的外观形状。网格渲染是将 3D 模型镂空，只留下模型表面的部分顶点，使模型成为线框物体。这就要求着色器需要丢弃一些片元，只保留部分片元，来绘制带有“洞”的物体。所以网格渲染的效果和其他渲染效果有很大的不同，被丢弃的片元并不会留下痕迹。那么如何才能丢弃片元呢？GLSL 中提供了 discard 命令，在像素着色器中使用 discard 命令可实现模型网格化的效果。Discard 命令会导致丢弃片元而不是使用片元来更新帧缓冲区，所以可以利用 discard 命令这一特性来绘制有“洞”的几何体，即模型的网格化。网格渲染的顶点着色器和条纹渲染的顶点着色器完全相同。

1. 算法步骤

1）顶点着色器

顶点着色器负责光照的计算，包括漫反射和镜面反射颜色值。

第一步：计算顶点眼睛坐标和法向量。

```
vec3 ecPosition     = vec3( gl_ModelViewMatrix * gl_Vertex );
vec3 tnorm          = normalize( gl_NormalMatrix * gl_Normal );
```

第二步：根据眼睛坐标计算光线矢量和查看方向矢量。

```
vec3  lightVec      = normalize( LightPosition - ecPosition );
vec3  viewVec       = normalize( EyePosition - ecPosition );
```

第三步：计算光线矢量和查看方向矢量的矢量和。

```
vec3 Hvec              = normalize( viewVec + lightVec );
```

第四步：利用光线矢量和查看方向矢量的矢量和与法向量求镜面反射强度，并限制在（0，1）之间，然后进行锐化。

```
float  spec   = clamp( dot( Hvec, tnorm ), 0.0, 1.0 );
Spec          = pow( spec, 16.0 );
```

第五步：计算漫反射和镜面反射颜色值。

```
DiffuseColor      = LightColor * vec3( Kd * dot( lightVec, tnorm ) );
DiffuseColor      = clamp( Ambient + DiffuseColor, 0.0, 1.0 );
SpecularColor     = clamp( ( LightColor * Specular * spec ), 0.0, 1.0 );
```

第六步：计算纹理坐标。像素着色器中绘制条纹颜色或背景颜色就是根据纹理坐标绘制的。

```
if( gl_MultiTexCoord0.s == 0 && gl_MultiTexCoord0.t == 0 )
  gl_TexCoord[0] = gl_Vertex;
else
  gl_TexCoord[0]   = gl_MultiTexCoord0;
```

第七步：这是每一个顶点着色器必须做的事情，计算齐次顶点位置。

```
gl_Position            = ftransform( );
```

2）像素着色器

在像素着色器中会使用 discard 命令来实现绘制模型的线框。discard 命令会导致丢弃片元而不是使用片元来更新帧缓冲区，以此来绘制带有“洞”几何体。

第一步：计算要丢弃的片元的纹理坐标值。

```
float ss      = fract( gl_TexCoord[0].s * Scale.s );
float tt      = fract( gl_TexCoord[0].t * Scale.t );
```

第二步：根据阈值判断是否丢弃，大于阈值则丢弃。

```
if ( ( ss > Threshold.s ) && ( tt > Threshold.t ) ) discard;
```

第三步：计算着色值。

```
vec3 finalColor      = SurfaceColor * DiffuseColor + SpecularColor;
```

第四步：计算内置变量 gl_FragtColor 的值，即最后输出的颜色。

```
gl_FragColor      = vec4( finalColor, 1.0 );
```

2. 参数界面

如图 8-13 所示是网格渲染的参数界面。其中参数有环境光、镜面反射、用于调整网格频率的缩放比例因子、判断是否丢弃片元的阈值。设置不同大小的阈值可以得到不同

的网格效果。

Ambient	(0.2, 0.2, 0.2)
EyePosition	(0, 0, 4)
Kd	0.8
LightColor	(0.9, 0.8, 0.7)
Scale	(10, 10)
Specular	(0.2, 0.2, 0.2)
SurfaceColor	(0.9, 0.7, 0.25)
Threshold	(0.13, 0.13)

图 8-13　网格渲染参数界面

3. 完整代码

1）顶点着色器

```
uniform vec3  LightPosition;
uniform vec3  LightColor;
uniform vec3  EyePosition;
uniform vec3  Specular;
uniform vec3  Ambient;
uniform float Kd;
varying vec3  DiffuseColor;
varying vec3  SpecularColor;

void main(void)
{
    vec3 ecPosition = vec3 (gl_ModelViewMatrix * gl_Vertex);
    vec3 tnorm      = normalize(gl_NormalMatrix * gl_Normal);
    vec3 lightVec   = normalize(LightPosition - ecPosition);
    vec3 viewVec    = normalize(EyePosition - ecPosition);
    vec3 Hvec       = normalize(viewVec + lightVec);
    float spec = abs(dot(Hvec,tnorm));
    spec = pow(spec,16.0);
    DiffuseColor    = LightColor * vec3 (Kd * abs(dot(lightVec,tnorm)));
    DiffuseColor    = clamp(Ambient + DiffuseColor,0.0,1.0);
    SpecularColor   = clamp((LightColor * Specular * spec),0.0,1.0);
    if(gl_MultiTexCoord0.s == 0&&gl_MultiTexCoord0.t == 0)
        gl_TexCoord[0] = gl_Vertex;
    else
        gl_TexCoord[0]  = gl_MultiTexCoord0;
    gl_Position     = ftransform();
}
```

2）像素着色器

```
varying vec3  DiffuseColor;
```

```
varying vec3  SpecularColor;
uniform vec2  Scale;
uniform vec2  Threshold;
uniform vec3  SurfaceColor;

void main( )
{
    float ss = fract( gl_TexCoord[ 0 ]. s * Scale. s) ;
    float tt = fract( gl_TexCoord[ 0 ]. t * Scale. t) ;

    if ( ( ss > Threshold. s)  &&  ( tt > Threshold. t) )  discard;

    vec3 finalColor = SurfaceColor * DiffuseColor + SpecularColor;
    gl_FragColor = vec4( finalColor,1. 0) ;
}
```

4. 效果图

如图 8-14 所示，第一行是兔子模型在不同参数下的网格渲染效果；第二行是不同三维模型的网格渲染效果。

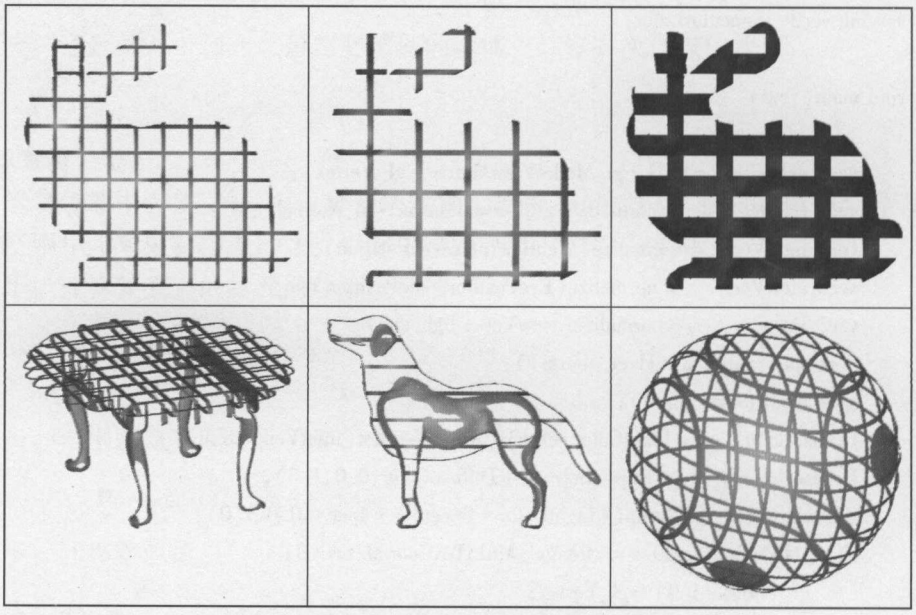


图 8-14　网格渲染效果

第 9 章 光照

9.1 半球光照

在真实世界中，一个物体只有反射光线或自己本身发光，才可以被看到。在计算机模拟中，三维物体也需要反射光线或自发光才能得到显示。OpenGL 本身内置的光照方式只是计算机对光照进行模拟的一种方式。使用 GLSL 语言进行编程就可以得到更多、更灵活的光照方式，从而渲染出更加逼真的光照效果。OpenGL 内置的光照是最成功的一种实时光照方法。但这种光照方法也具有一些缺点。在真实的世界中，一个物体除了受到光源的照射之外，还受到其他物体反射的光照。这种物体之间反射的光线是视觉上感受真实场景很重要的一部分。OpenGL 内置的光照方法采用的是用环境光来模拟物体之间的反射光照。虽然环境光方法有一定的效果，但由于环境光对所有的物体都是一样的，因此对没有被光源直接照射的部分呈现出一种平板、不真实的视觉效果。在真实世界中，光源通常不是点光源和聚光灯光源，而是一个区域光源，也就是光源本身有一定的体积和面积。例如，通过窗户照射到地板的间接光源，此处窗户可以看作一个具有一定面积的光源。

在看不到太阳的有云的天气里，整个天空都可以看作一个形状为半球的光源。这种光照效果称为是半球光照（Hemisphere Lighting）。半球光照的基本原理是把整个光照分为两个半球，上半球代表天空，下半球代表地面，如图 9-1 所示。

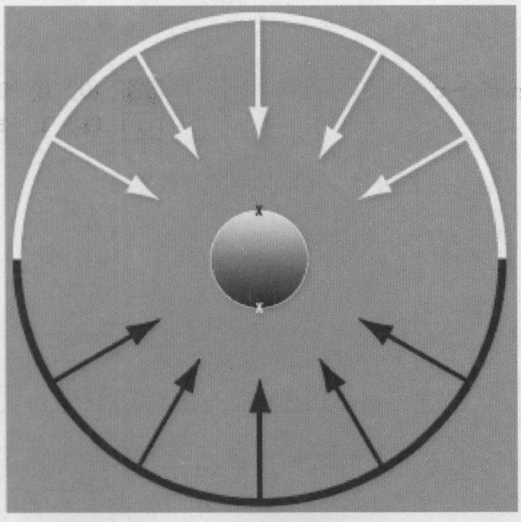


图 9-1 半球光照

1. 算法步骤

1）顶点着色器

第一步：计算眼睛坐标。

```
vec3 ecPosition        = vec3(gl_ModelViewMatrix * gl_Vertex);
```

第二步：计算法向量。

```
vec3 tnorm             = normalize(gl_NormalMatrix * gl_Normal);
```

第三步：计算光线矢量。

```
vec3 lightVec          = normalize(LightPosition - ecPosition);
```

第四步：计算漫反射强度。

```
float costheta         = dot(tnorm,lightVec);
float a                = 0.5 + 0.5 * costheta;
```

第五步：计算漫反射颜色值。根据从模型上方和下方投射的颜色和漫反射强度计算漫反射颜色值。

```
DiffuseColor           = mix(GroundColor,SkyColor,a);
```

第六步：计算齐次顶点位置。

```
gl_Position            = ftransform();
```

2）像素着色器

像素着色器只需根据漫反射颜色找色即可。

```
gl_FragColor    = vec4(DiffuseColor,1.0);
```

2. 参数界面

如图 9-2 所示是半球光照的参数界面。其中参数有从模型上方投射的光照和从模型下方投射的光照。设置的光照颜色不同可以得到不同的渲染效果。

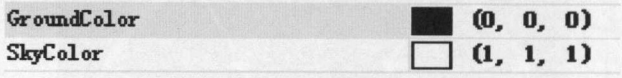


图 9-2　半球光照的参数界面

3. 完整代码

1）顶点着色器

```
uniform vec3 LightPosition;
uniform vec3 SkyColor;
uniform vec3 GroundColor;
varying vec3  DiffuseColor;

void main(void)
```

```
{
    vec3 ecPosition = vec3( gl_ModelViewMatrix * gl_Vertex) ;
    vec3 tnorm      = normalize( gl_NormalMatrix * gl_Normal) ;
    vec3 lightVec   = normalize( LightPosition - ecPosition) ;
    float costheta  = dot( tnorm, lightVec) ;
    float a         = 0. 5 + 0. 5 * costheta;
    DiffuseColor    = mix( GroundColor, SkyColor, a) ;
    gl_Position     = ftransform( ) ;
}
```

2）像素着色器

```
varying vec3  DiffuseColor;

void main( void)
{
    gl_FragColor = vec4( DiffuseColor, 1. 0) ;
}
```

4. 效果图

如图9-3所示，第一行是兔子三维模型在不同参数下的半球光照渲染效果；第二行是不同的三维模型半球光照渲染效果。从中可以看出，三维模型用半球光照可以渲染出比较真实的效果。

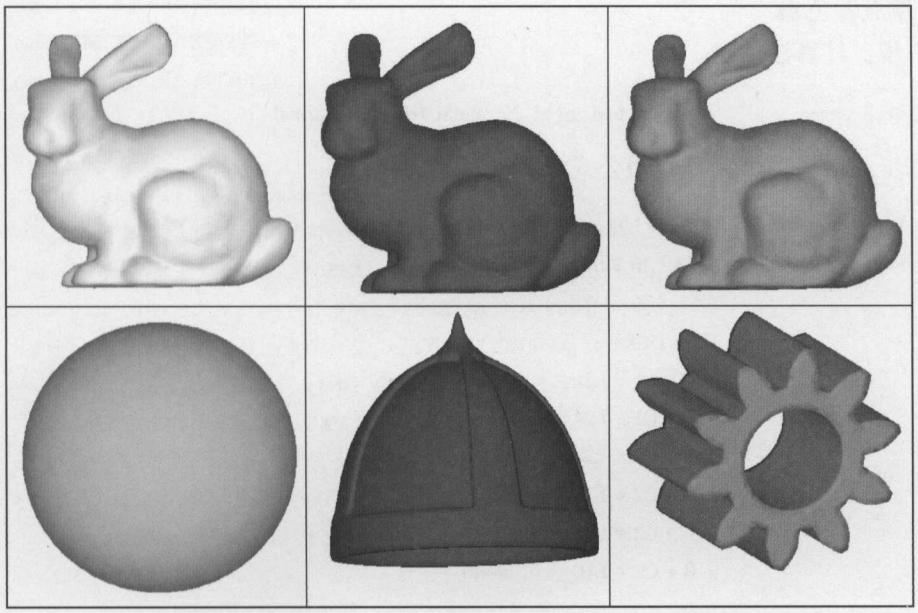


图9-3　半球光照渲染效果

9.2 球形调和光照

球形调和光照（Spherical Harmonics）是实时渲染技术中的一种，属于 Precompute Radiance Transfer（PRT）的范畴。经过预处理并存储相应的信息之后，它可以产生高质量的渲染及阴影效果。球形光照需要使用新的光照方程来代替通常的光照方程，并将该方程中的相关信息使用球形基函数来投影到频度空间，并用系数进行表示（该变换与信号处理中的 Fourier 变换的道理相同）以一定的方式存储。在渲染的过程中，就结合球谐变换的特性及这些预先存储的系数信息来对原始的光照方程进行还原并对场景进行着色。这个过程是对无限积分进行有限近似的过程，但通常可以得到较为理想的效果。当然，球形函数也具有很多神奇的特性（如旋转不变性，Rotation Invariant），这些特性就为其在渲染及其他领域中的应用又带来很多优势。之前关注的三维模型检索及特征值的提取就同样可以借助于球形变换来对 3D 模型进行基于内容的特征分析与变换（3D Model & Spherical Harmonic & Moments），效果也非常好。一般来说，使用球形变换的特性可以得到一定程度上动态的光照渲染效果，如场景或环境光的旋转等。球形调和光照的计算公式如下。

$$\text{Diffuse} = C_1 L_{22}(x^2 - y^2) + C_3 L_{20} z^2 + C_4 L_{00} - C_5 L_{20} + 2C_1(L_{2m2}xy + L_{21}xz - L_{2m1}yz) + 2C_2(L_{11}x + L_{1m1}y - L_{10}z)$$

$C_1 \sim C_5$ 是根据根据这个公式推导出来的，9 个 L 系数是 9 个调和基函数系数，(x,y,z) 是顶点的归一化法线向量。

1. 算法步骤

1）顶点着色器

第一步：计算法向量。

```
vec3 tnorm          = normalize( gl_NormalMatrix * gl_Normal );
```

第二步：根据球形调和光照公式计算漫反射颜色。

```
DiffuseColor    = C1 * L22 * ( tnorm. x * tnorm. x - tnorm. y * tnorm. y ) +
                  C3 * L20 * tnorm. z * tnorm. z +
                  C4 * L00 -
                  C5 * L20 +
                  2. 0 * C1 * L2m2 * tnorm. x * tnorm. y +
                  2. 0 * C1 * L21   * tnorm. x * tnorm. z +
                  2. 0 * C1 * L2m1 * tnorm. y * tnorm. z +
                  2. 0 * C2 * L11   * tnorm. x +
                  2. 0 * C2 * L1m1 * tnorm. y +
                  2. 0 * C2 * L10   * tnorm. z;
```

第三步：计算调整后的漫反射颜色。ScaleFactor 是可供用户调整的参数因子，调整因子的大小可得到不同的漫反射颜色。

```
DiffuseColor        * = ScaleFactor;
```

第四步：计算顶点的齐次坐标。

```
gl_Position          = ftransform( );
```

2）像素着色器

像素着色器使用非常简单的着色模式，利用顶点着色器内计算的漫反射颜色值，直接复制给像素着色器即可。

```
gl_FragColor    = vec4( DiffuseColor,1.0 );
```

2. 参数界面

如图9-4所示是半球光照的参数界面。ScaleFactor是可供用户调整的参数因子，调整因子的大小可得到不同的漫反射颜色。

图9-4　半球光照渲染参数界面

3. 完整代码

1）顶点着色器

```
varying vec3   DiffuseColor;
uniform float ScaleFactor;

const float C1 = 0.429043;
const float C2 = 0.511664;
const float C3 = 0.743125;
const float C4 = 0.886227;
const float C5 = 0.247708;

#if 1
// Constants for Grace Cathedral lighting
const vec3 L00   = vec3( 0.78908 ,0.43710 ,0.54161 );
const vec3 L1m1 = vec3( 0.39499 ,0.34989 ,0.60488 );
const vec3 L10   = vec3( -0.33974 , -0.18236 , -0.26940 );
const vec3 L11   = vec3( -0.29213 , -0.05562 ,0.00944 );
const vec3 L2m2 = vec3( -0.11141 , -0.05090 , -0.12231 );
const vec3 L2m1 = vec3( -0.26240 , -0.22401 , -0.47479 );
const vec3 L20   = vec3( -0.15570 , -0.09471 , -0.14733 );
const vec3 L21   = vec3( 0.56014 ,0.21444 ,0.13915 );
const vec3 L22   = vec3( 0.21205 , -0.05432 , -0.30374 );
#endif

#if 0
// Constants for Eucalyptus Grove lighting
const vec3 L00   = vec3( 0.3783264 ,0.4260425 ,0.4504587 );
```

```
const vec3 L1m1 = vec3( 0.2887813,0.3586803,0.4147053);
const vec3 L10   = vec3( 0.0379030,0.0295216,0.0098567);
const vec3 L11   = vec3( -0.1033028, -0.1031690, -0.0884924);
const vec3 L2m2 = vec3( -0.0621750, -0.0554432, -0.0396779);
const vec3 L2m1 = vec3( 0.0077820, -0.0148312, -0.0471301);
const vec3 L20   = vec3( -0.0935561, -0.1254260, -0.1525629);
const vec3 L21   = vec3( -0.0572703, -0.0502192, -0.0363410);
const vec3 L22   = vec3( 0.0203348, -0.0044201, -0.0452180);
#endif

#if 0
// Constants for St. Peter's Basilica lighting
const vec3 L00   = vec3( 0.3623915,0.2624130,0.2326261);
const vec3 L1m1 = vec3( 0.1759130,0.1436267,0.1260569);
const vec3 L10   = vec3( -0.0247311, -0.0101253, -0.0010745);
const vec3 L11   = vec3( 0.0346500,0.0223184,0.0101350);
const vec3 L2m2 = vec3( 0.0198140,0.0144073,0.0043987);
const vec3 L2m1 = vec3( -0.0469596, -0.0254485, -0.0117786);
const vec3 L20   = vec3( -0.0898667, -0.0760911, -0.0740964);
const vec3 L21   = vec3( 0.0050194,0.0038841,0.0001374);
const vec3 L22   = vec3( -0.0818750, -0.0321501,0.0033399);
#endif

#if 0
// Constants for Uffizi Gallery lighting
const vec3 L00   = vec3(0.3168843,0.3073441,0.3495361);
const vec3 L1m1 = vec3(0.3711289,0.3682168,0.4292092);
const vec3 L10   = vec3( -0.0034406, -0.0031891, -0.0039797);
const vec3 L11   = vec3( -0.0084237, -0.0087049, -0.0116718);
const vec3 L2m2 = vec3( -0.0190313, -0.0192164, -0.0250836);
const vec3 L2m1 = vec3( -0.0110002, -0.0102972, -0.0119522);
const vec3 L20   = vec3( -0.2787319, -0.2752035, -0.3184335);
const vec3 L21   = vec3(0.0011448,0.0009613,0.0008975);
const vec3 L22   = vec3( -0.2419374, -0.2410955, -0.2842899);
#endif

#if 0
// Constants for Galileo's tomb lighting
const vec3 L00   = vec3(1.0351604,0.7603549,0.7074635);
const vec3 L1m1 = vec3(0.4442150,0.3430402,0.3403777);
const vec3 L10   = vec3( -0.2247797, -0.1828517, -0.1705181);
const vec3 L11   = vec3(0.7110400,0.5423169,0.5587956);
```

```
const vec3 L2m2 = vec3(0.6430452,0.4971454,0.5156357);
const vec3 L2m1 = vec3(-0.1150112,-0.0936603,-0.0839287);
const vec3 L20  = vec3(-0.3742487,-0.2755962,-0.2875017);
const vec3 L21  = vec3(-0.1694954,-0.1343096,-0.1335315);
const vec3 L22  = vec3(0.5515260,0.4222179,0.4162488);
#endif

#if 0
// Constants for Vine Street kitchen lighting
const vec3 L00  = vec3(0.6396604,0.6740969,0.7286833);
const vec3 L1m1 = vec3(0.2828940,0.3159227,0.3313502);
const vec3 L10  = vec3(0.4200835,0.5994586,0.7748295);
const vec3 L11  = vec3(-0.0474917,-0.0372616,-0.0199377);
const vec3 L2m2 = vec3(-0.0984616,-0.0765437,-0.0509038);
const vec3 L2m1 = vec3(0.2496256,0.3935312,0.5333141);
const vec3 L20  = vec3(0.3813504,0.5424832,0.7141644);
const vec3 L21  = vec3(0.0583734,0.0066377,-0.0234326);
const vec3 L22  = vec3(-0.0325933,-0.0239167,-0.0330796);
#endif

#if 0
// Constants for Breezeway lighting
const vec3 L00  = vec3(0.3175995,0.3571678,0.3784286);
const vec3 L1m1 = vec3(0.3655063,0.4121290,0.4490332);
const vec3 L10  = vec3(-0.0071628,-0.0123780,-0.0146215);
const vec3 L11  = vec3(-0.1047419,-0.1183074,-0.1260049);
const vec3 L2m2 = vec3(-0.1304345,-0.1507366,-0.1702497);
const vec3 L2m1 = vec3(-0.0098978,-0.0155750,-0.0178279);
const vec3 L20  = vec3(-0.0704158,-0.0762753,-0.0865235);
const vec3 L21  = vec3(0.0242531,0.0279176,0.0335200);
const vec3 L22  = vec3(-0.2858534,-0.3235718,-0.3586478);
#endif

#if 0
// Constants for Campus Sunset lighting
const vec3 L00  = vec3(0.7870665,0.9379944,0.9799986);
const vec3 L1m1 = vec3(0.4376419,0.5579443,0.7024107);
const vec3 L10  = vec3(-0.1020717,-0.1824865,-0.2749662);
const vec3 L11  = vec3(0.4543814,0.3750162,0.1968642);
const vec3 L2m2 = vec3(0.1841687,0.1396696,0.0491580);
const vec3 L2m1 = vec3(-0.1417495,-0.2186370,-0.3132702);
const vec3 L20  = vec3(-0.3890121,-0.4033574,-0.3639718);
```

```
const vec3 L21   = vec3( 0.0872238,0.0744587,0.0353051);
const vec3 L22   = vec3( 0.6662600,0.6706794,0.5246173);
#endif

#if 0
// Constants for Funston Beach Sunset lighting
const vec3 L00   = vec3(0.6841148,0.6929004,0.7069543);
const vec3 L1m1 = vec3(0.3173355,0.3694407,0.4406839);
const vec3 L10   = vec3( -0.1747193, -0.1737154, -0.1657420);
const vec3 L11   = vec3( -0.4496467, -0.4155184, -0.3416573);
const vec3 L2m2 = vec3( -0.1690202, -0.1703022, -0.1525870);
const vec3 L2m1 = vec3( -0.0837808, -0.0940454, -0.1027518);
const vec3 L20   = vec3( -0.0319670, -0.0214051, -0.0147691);
const vec3 L21   = vec3(0.1641816,0.1377558,0.1010403);
const vec3 L22   = vec3(0.3697189,0.3097930,0.2029923);
#endif

#if 0
// Constants for Old Town Square lighting
const vec3 L00   = vec3(0.871297,0.875222,0.864470);
const vec3 L1m1 = vec3(0.175058,0.245335,0.312891);
const vec3 L10   = vec3(0.034675,0.036107,0.037362);
const vec3 L11   = vec3( -0.004629, -0.029448, -0.048028);
const vec3 L2m2 = vec3( -0.120535, -0.121160, -0.117507);
const vec3 L2m1 = vec3(0.003242,0.003624,0.007511);
const vec3 L20   = vec3( -0.028667, -0.024926, -0.020998);
const vec3 L21   = vec3( -0.077539, -0.086325, -0.091591);
const vec3 L22   = vec3( -0.161784, -0.191783, -0.219152);
#endif

void main(void)
{
    vec3 tnorm       = normalize(gl_NormalMatrix * gl_Normal);
    DiffuseColor     = C1 * L22 * (tnorm.x * tnorm.x - tnorm.y * tnorm.y) +
                       C3 * L20 * tnorm.z * tnorm.z +
                       C4 * L00 -
                       C5 * L20 +
                       2.0 * C1 * L2m2 * tnorm.x * tnorm.y +
                       2.0 * C1 * L21  * tnorm.x * tnorm.z +
                       2.0 * C1 * L2m1 * tnorm.y * tnorm.z +
                       2.0 * C2 * L11  * tnorm.x +
                       2.0 * C2 * L1m1 * tnorm.y +
```

```
                    2.0 * C2 * L10    * tnorm.z;
    DiffuseColor    * = ScaleFactor;
    gl_Position     = ftransform();
}
```

2）像素着色器

```
varying vec3  DiffuseColor;
void main(void)
{
    gl_FragColor = vec4(DiffuseColor,1.0);
}
```

4. 效果图

如图9-5所示是球形调和光照的效果图，从中可以看出，球形调和光照能够比较真实的反应光照的柔和感。第一行是同样的三维模型在不同光照参数下的渲染效果；第二行是不同的三维模型的渲染效果。

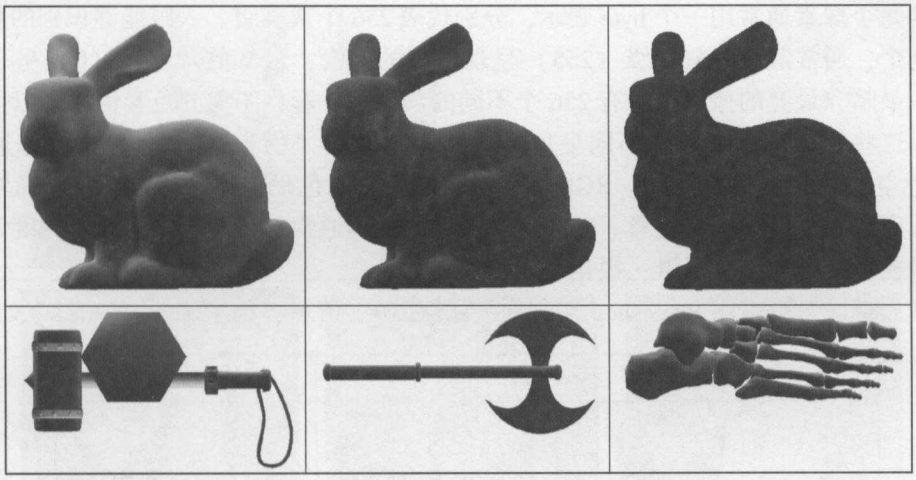


图9-5　球形调和照效果

第 10 章 图像处理

10.1 概述

图像处理是指对图像进行分析、加工和处理，使其满足视觉、心理及其他要求的技术。目前大多数的图像是以数字形式存储的，因此图像处理在很多情况下是指数字图像处理，计算机图像大致可以分为：二值图像、灰度图像和彩色图像三种。二值图像中，每个像素要么是黑，要么是白，理论上二值图像的每个像素只需要一个 bit 表示：0 表示黑，1 表示白。灰度图像的每个像素通常用一个 byte 表示，分别代表 256 个灰度级。人眼能够识别的灰度级大约是 100 个，通常最高的灰度级（255）呈现最亮的像素，最低的灰度级（0）呈现最暗的像素，在最暗和最亮的像素之间有 256 个不同的灰度级。彩色图像的色彩信息可以用多种方式呈现，这些表示彩色图像的不同呈现方式称为图像的“色彩空间”，彩色图像通常使用 RGB 彩色空间和 HSL 彩色空间。RGB 彩色空间使用三原色呈现图像色彩。HSL 通常使用色相、饱和度和亮度呈现图像色彩。基于数字图像的处理通常有对比度的调节、亮度调节、颜色空间转换、边缘检测、平滑、锐化等操作。

图像处理通常可以用 Photoshop 来实现，如图 10-1 所示各种图像的 Photoshop 处理结果。

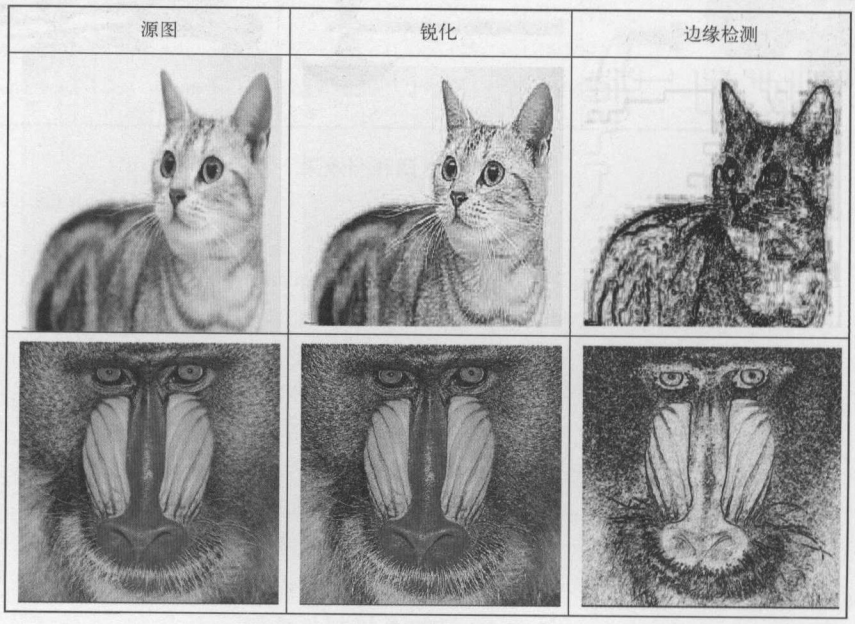


图 10-1　Photoshop 处理结果

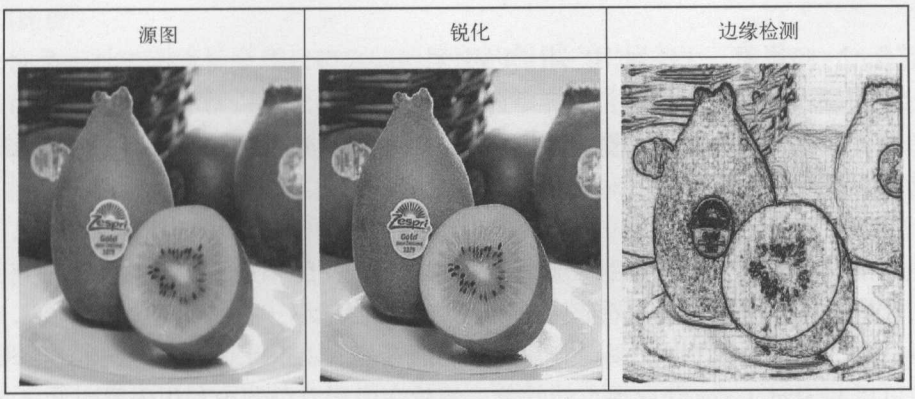


图 10-1　Photoshop 处理结果（续）

但也可以用 GPU 来处理图像，如图 10-2 所示。

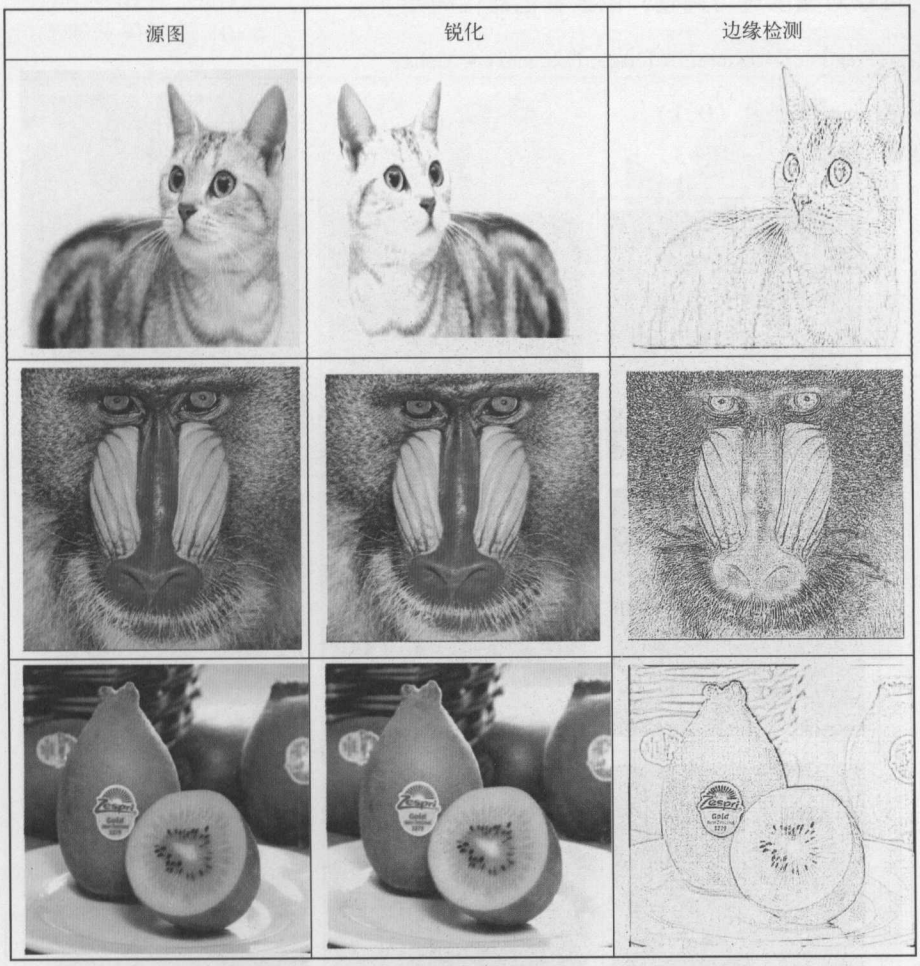


图 10-2　GPU 处理结果

10.2 亮度、对比度和饱和度

在图像处理中经常会遇到调整图像的明暗、调整图像颜色的差异及调整图像色彩的鲜艳程度，这就会涉及图像的亮度、对比度、饱和度。

1. 亮度

亮度是指画面的明亮程度，图像颜色值的综合明亮程度，且与颜色的色调无关，亮度值对应为颜色值。亮度值越大图像看起来越明亮，亮度值越小，图像看起来越暗淡。

在 OpenGL 中，实现亮度的调整非常简单。内置 gl_Color 可获取顶点的颜色，即在顶点着色器中可以利用 gl_Color 获取顶点的颜色，然后在像素着色器中利用内置变量 gl_FragColor 计算最终的着色值，代码如下。

```
gl_FragColor = gl_Color
```

我们可以对亮度进行调整，像素着色器代码如下。

```
gl_FragColor = texture2D( Image , TexCoord ) * Alpha;
```

其中 Alpha 取值为（0,1）。

亮度调整效果见图 10-3。

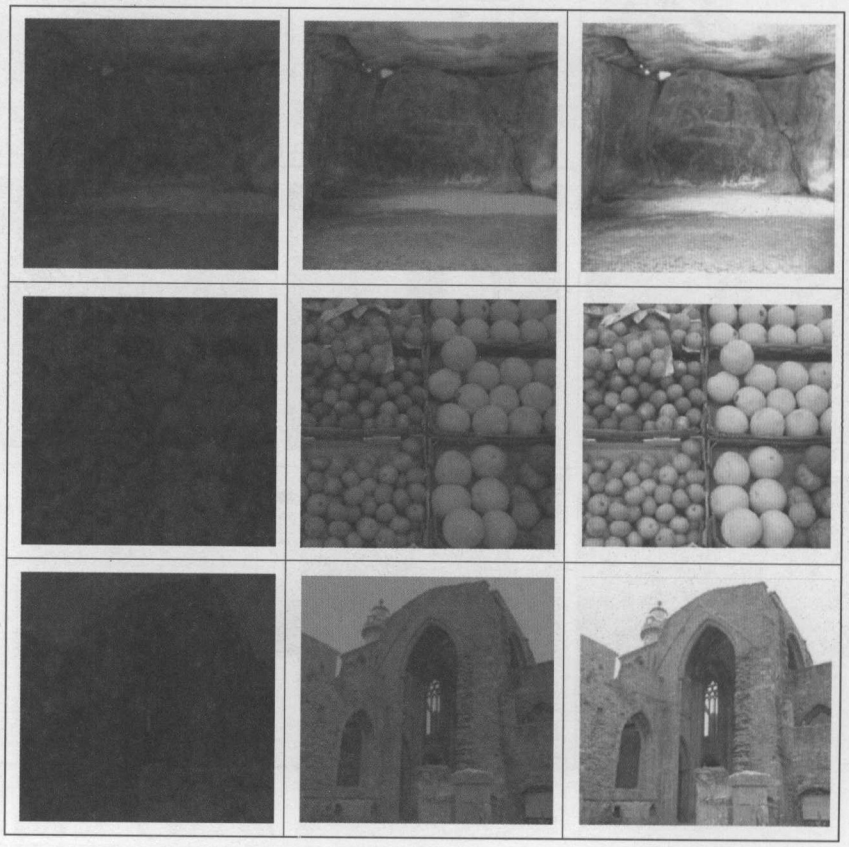


图 10-3　亮度调整效果

2. 对比度

对比度是指投影图像最亮和最暗之间的区域的比率，比值越大，从黑到白的渐变层次就越多，从而色彩表现越丰富。对比度对视觉效果的影响非常关键，一般来说对比度越大，图像越清晰醒目，色彩也越鲜明艳丽；而对比度小，则会让整个画面都灰蒙蒙的。高对比度对于图像的清晰度、细节表现、灰度层次表现都有很大帮助。对比度越高图像效果越好，色彩会更饱和，反之，对比度低则画面会显得模糊，色彩也不鲜明。

通过提高对比度可以使图像变得更加清晰，图像对比度的调整是在图像亮度的基础上进行操纵的，所以在顶点着色器中依然用内置 gl_Color 可获取顶点的颜色，在像素着色器中一般用所得到的顶点亮度值和已知给定亮度值的线性组合来表示，像素着色器代码如下。

```
vec3 color      = texture2D( Image , TexCoord ). rgb;
color           = mix( AvgLuminance , color , Alpha ) ;
gl_FragColor    = vec4( color , 1. 0 ) ;
```

其中 Alpha 取值为（0,1）。

对比度调整效果见图 10-4。

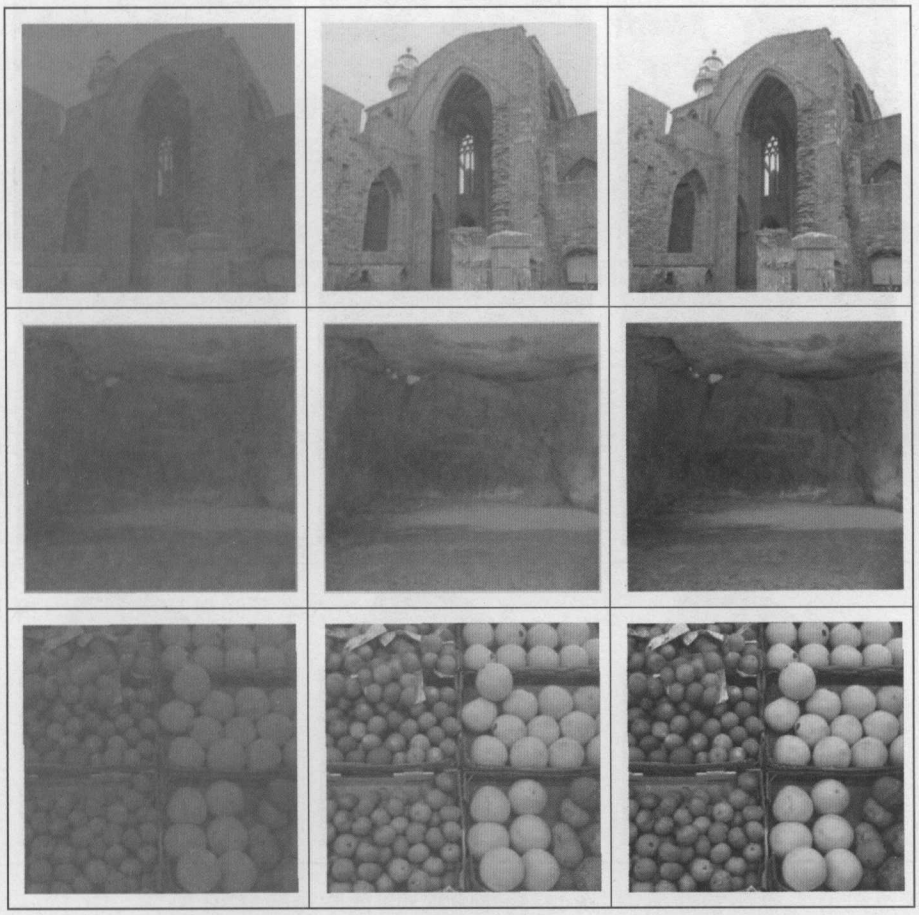


图 10-4　对比度效果

3. 饱和度

饱和度其实是指色彩的纯度，纯度越高，表现越鲜明，纯度越低，表现则越黯淡，色饱和度表示光线的彩色深浅度或鲜艳度，取决于彩色中的白色光含量，白光含量越高，即彩色光含量就越低，色彩饱和度即越低，反之亦然。其数值为百分比，介于0% ~100% 之间。纯白光的色彩饱和度为0%，而纯彩色光的饱和度则为100%。

饱和度的调整和对比度调整类似。在顶点着色器中用 gl_ Color 来获取顶点的颜色，在像素着色器中用于调整饱和度的做法是利用线性组合法，不过除了顶点的颜色值外，另一个参数是顶点颜色值和给定已知颜色值的点积。像素着色器代码如下。

```
vec3 texColor    = texture2D( Image, TexCoord). rgb;
vec3 intensity   = vec3( dot( texColor, LumCoeff) );
vec3 color       = mix( intensity, texColor, Alpha);
gl_FragColor     = vec4( color, 1. 0);
```

饱和度调整效果见图 10-5。

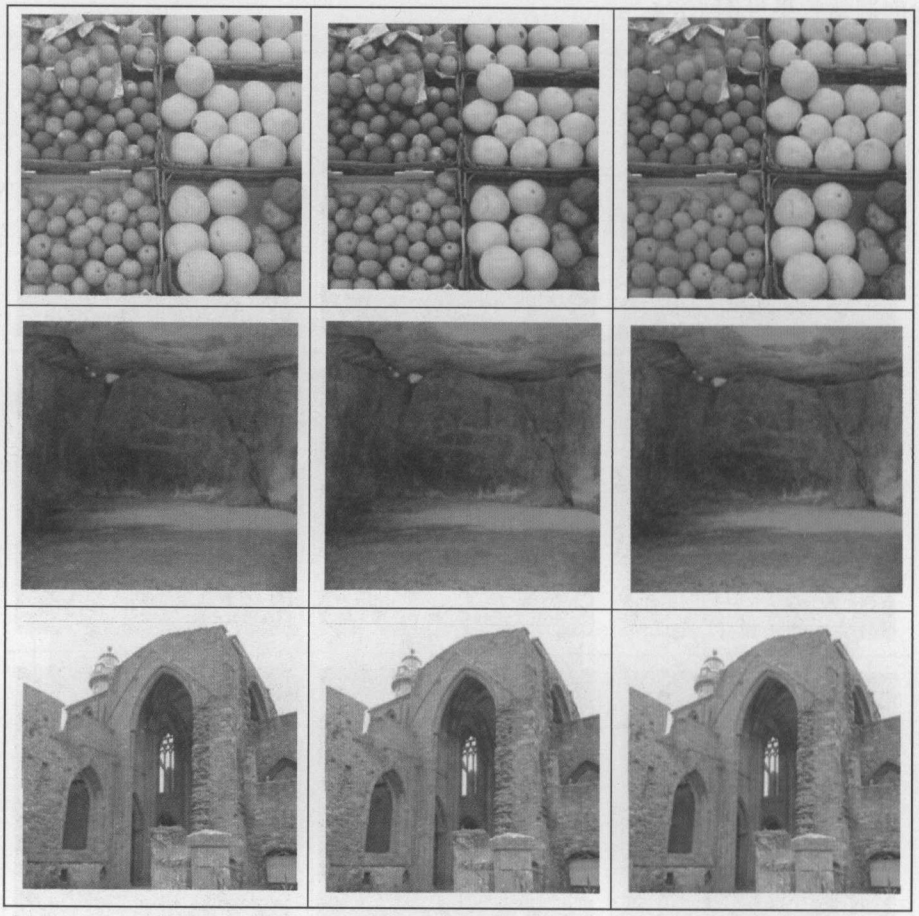


图 10-5　饱和度调整效果

10.3　颜色空间转换

10.3.1　介绍

所谓颜色模型就是指某个三维颜色空间中的一个可见光子集，它包含某个颜色域的所有颜色。例如，RGB 颜色模型就是三维直角坐标颜色系统的一个单位正方体。颜色模型的用途是在某个颜色域内方便地指定颜色，由于每一个颜色域都是可见光的子集，所以任何一个颜色模型都无法包含所有的可见光。在大多数的彩色图形显示设备一般都使用红、绿、蓝三原色，我们的真实感图形学中主要的颜色模型也是 RGB（红绿蓝）模型，但红、绿、蓝颜色模型用起来不太方便，它与直观的颜色概念如色调、饱和度和亮度等没有直接的联系。颜色空间已经提出上百种，大部分只是局部的改变或专用于某一领域，常用有 RGB、CMY、CIE 等。

RGB 是依据人眼识别的颜色定义出的空间，可表示大部分颜色。在计算机中编程 RGB 每一个分量值都用 8 位（bit）表示，可以产生 256 × 256 × 256 = 16 777 216 中颜色，这就是经常所说的“24 位真彩色”。但在科学研究中一般不采用 RGB 颜色空间，因为它的细节难以进行数字化的调整。它将色调，亮度，饱和度三个量放在一起表示，很难分开。它是最通用的面向硬件的彩色模型。该模型用于彩色监视器和一大类彩色视频摄像。

CMY 是工业印刷采用的颜色空间，它与 RGB 对应。简单的类比 RGB 来源于是物体发光，而 CMY 是依据反射光得到的。具体应用如打印机：一般采用四色墨盒，即 CMY 加黑色墨盒。相比于 RGB，CMY（CMYK）颜色空间是另一种基于颜色减法混色原理的颜色模型。在工业印刷中，它描述的是需要在白色介质上使用何种油墨，通过光的反射显示出颜色的模型。

CIE 是由国际照明委员会于 1931 年定义的，通常作为国际性的颜色空间标准，用作颜色的基本度量方法。它根据人类对颜色的感觉定义了一个与设备无关的颜色空间，在科学计算中得到了广泛的应用。对不能直接相互转换的两个颜色空间，可以利用这类颜色空间作为过渡性的颜色空间。

10.3.2　RGB 和 CMY 相互转换

RGB 和 CMY 相互转换的数学公式为：

$$\begin{bmatrix} C \\ M \\ Y \end{bmatrix} = \begin{bmatrix} 1-R \\ 1-G \\ 1-B \end{bmatrix}$$

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1-C \\ 1-M \\ 1-Y \end{bmatrix}$$

在 OpenGL 中，实现 RGB 颜色空间到 CMY 颜色空间的转换非常简单，在顶点着色器内只需读入纹理的坐标，颜色的转换在像素着色器内执行，读取到纹理上的颜色值后根据相应的转换公式就能完成相应的转换。以下是转换的算法步骤。

1. RGB 转换为 CMY

1）顶点着色器

第一步：计算顶点位置。

```
gl_Position = ftransform();
```

第二步：计算纹理坐标。这里用内置的变量 gl_MultiTexCoord0 获取纹理单元上的纹理坐标并存储在内置易变变量 gl_TexCoord[0]中，是像素着色器获取颜色值的基础。

```
gl_TexCoord[0] = gl_MultiTexCoord0.st;
```

2）像素着色器

第一步：根据获取纹理坐标获取纹理颜色值，BaseImage 是纹理图像。

```
vec3 rgbcolor = texture2D(BaseImage,TexCoord).rgb;
```

第二步：根据获取的纹理颜色值和转换公式计算 CMY 分量值。

```
float C = 1 - rgbcolor.r;
float m = 1 - rgbcolor.g;
float y = 1 - rgbcolor.b;
```

第三步：计算转换后的纹理颜色值，并存储在 gl_FragtColor 中。

```
vec3 Cmycolor  = vec3(c,m,y);
gl_FragColor   = vec4(Cmycolor,1);
```

2. CMY 转换为 RGB

从 CMY 转换到 RGB 仍然是用同样的思路，根据 CMY 值求出 RGB 值即可，以下是转换代码。

```
float r = 1 - C;
float g = 1 - m;
float b = 1 - y;
vec3 RgbColor = vec3(r,g,b);
gl_FragColor  = vec4(RgbColor,1);
```

完整代码如下。

（1）顶点着色器

```
varying vec2 TexCoord;
void main()

{
    gl_Position    = ftransform();
    gl_TexCoord[0] = gl_MultiTexCoord0;
}
```

（2）像素着色器

```
uniform sampler2D BaseImage;
uniform float Mode;
varying vec2 TexCoord;
void rgbtoCMY(vec3 color)
{
    float  c = 1 - color.r;
    float m = 1 - color.g;
    float  y = 1 - color.b;
    vec3 Cmycolor = vec3(c,m,y);
    gl_FragColor   = vec4(Cmycolor,1);
}
void main()
{
    vec3 rgbcolor = texture2D(BaseImage,TexCoord).rgb;
    float C = 1 - rgbcolor.r;
    float m = 1 - rgbcolor.g;
    float  y = 1 - rgbcolor.b;
      rgbtoCMY(rgbcolor);
}
```

3. 效果图

如图 10-6 所示是几幅图片在 RGB 和 CMY 颜色空间的效果，从图中可以看出不同颜色空间的特点。

10.3.3　RGB 和 CIE 相互转换

RGB 和 CIE 的转换用到 3 × 3 的矩阵，由 RGB 转换到 CIE 的转换矩阵为：

$$\begin{bmatrix} 0.412453, 0.212671, 0.019334 \\ 0.357580, 0.715160, 0.119193 \\ 0.180423, 0.072169, 0.950227 \end{bmatrix}$$

由 CIE 转换到 RGB 的转换矩阵为：

$$\begin{bmatrix} 3.240479, -0.969256, 0.055648 \\ -1.537150, 1.875992, -0.204043 \\ -0.498535, 0.041556, 1.057311 \end{bmatrix}$$

RGB 和 CIE 颜色空间的转换和 RGB 和 CMY 颜色空间的转换原理是一样的，只不过运用的数学公式不一样。在 OpenGL 中，顶点着色器负责读取纹理坐标，像素着色器负责执行颜色的转换。以下是转换的算法步骤。

1. RGB 转换为 CIE

1）顶点着色器

第一步：计算顶点位置。

```
gl_Position        = ftransform();
```

第二步：计算纹理坐标。这里用内置的变量 gl_MultiTexCoord0 获取纹理单元上的纹理坐标并存储在内置易变变量 gl_TexCoord[0]中，是像素着色器获取颜色值的基础。

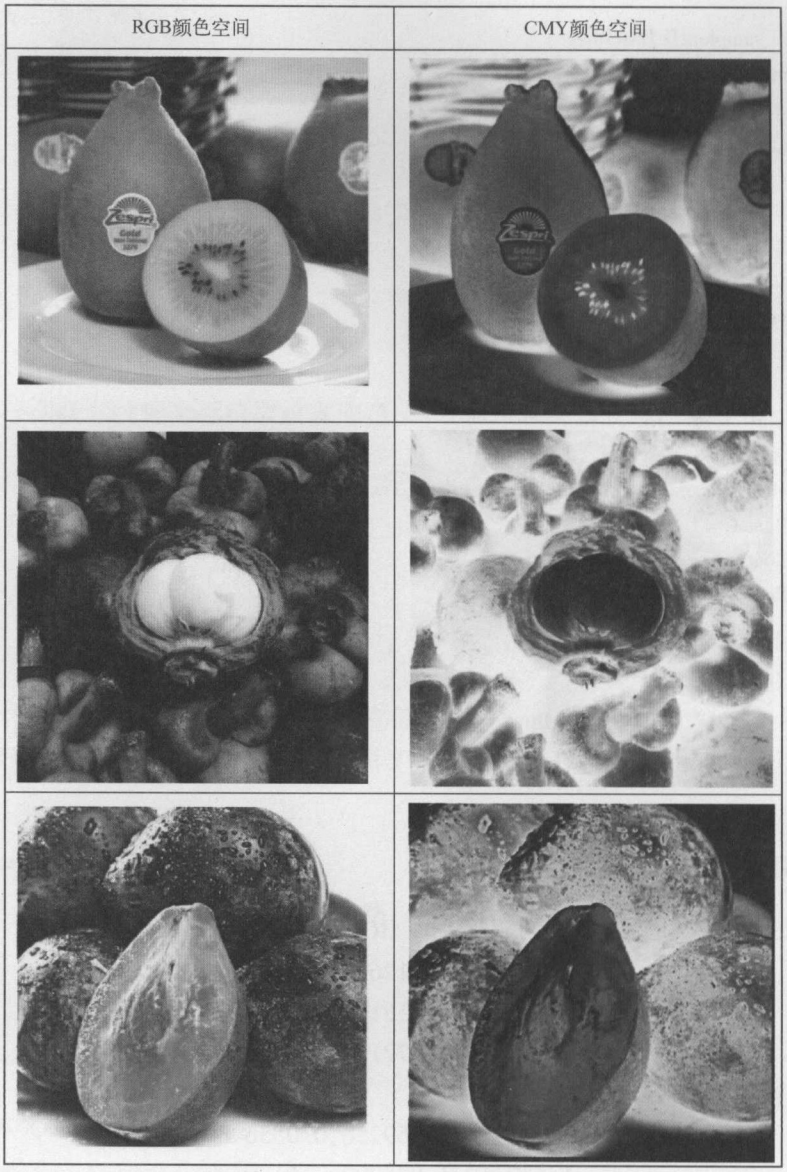


图 10-6　RGB 和 CMY 颜色空间的效果

```
gl_Position        = ftransform( );
```

2）像素着色器

第一步：根据获取纹理坐标获取纹理颜色值，BaseImage 是纹理图像。

```
vec3 rgbcolor      = texture2D( BaseImage, TexCoord). rgb;
```

第二步：根据获取的纹理颜色值和转换公式计算 CIE 分量值。

```
float c    = rgbcolor. r * 0. 412453 + rgbcolor. g * 0. 357580 + rgbcolor. b * 0. 180423;
float i    = rgbcolor. r * 0. 212671 + rgbcolor. g * 0. 715160 + rgbcolor. b * 0. 072169;
float e    = rgbcolor. r * 0. 019334 + rgbcolor. g * 0. 119193 + rgbcolor. b * 0. 950227;
```

第三步：计算转换后的纹理颜色值，并存储在 gl_FragtColor 中。

```
vec3 CieColor      = vec3(c,i,e);
gl_FragColor       = vec4(CieColor,1);
```

2. CIE 转换为 RGB

从 CIE 转换到 RGB 仍然是用同样的思路，根据 CIE 值求出 RGB 值即可，以下是转换代码。

```
float r = c * 3.240479 + i * -1.537150 + e * -0.498535;
float g = c * -0.969256 + i * 1.875992 + e * 0.041556;
float b = c * 0.055648 + i * -0.204043 + e * 1.057311;
vec3 RGBcolor = vec3(r,g,b);
gl_FragColor       = vec4(RGBColor,1);
```

完整代码如下。

(1) 顶点着色器

```
varying vec2 TexCoord;
void main()

{
    gl_Position      = ftransform();
    gl_TexCoord[0] = gl_MultiTexCoord0;
}
```

(2) 像素着色器

```
uniform sampler2D BaseImage;
uniform float Mode;
varying vec2 TexCoord;
void rgbtoCIE(vec3 color)
{
    float c = color.r * 0.412453 + color.g * 0.357580 + color.b * 0.180423;
    float i = color.r * 0.212671 + color.g * 0.715160 + color.b * 0.072169;
    float e = color.r * 0.019334 + color.g * 0.119193 + color.b * 0.950227;
    vec3 CieColor = vec3(c,i,e);
    gl_FragColor = vec4(CieColor,1);
}
void main()
{
    vec3 rgbcolor = texture2D(BaseImage,TexCoord).rgb;
    float c = rgbcolor.r * 0.412453 + rgbcolor.g * 0.357580 + rgbcolor.b * 0.180423;
    float i = rgbcolor.r * 0.212671 + rgbcolor.g * 0.715160 + rgbcolor.b * 0.072169;
    float e = rgbcolor.r * 0.019334 + rgbcolor.g * 0.119193 + rgbcolor.b * 0.950227;

    rgbtoCIE(rgbcolor);
}
```

3. 效果图

如图 10-7 所示是几幅图片在 RGB 和 CIE 颜色空间的效果，从中可以看出，两个空间的不同和特点。

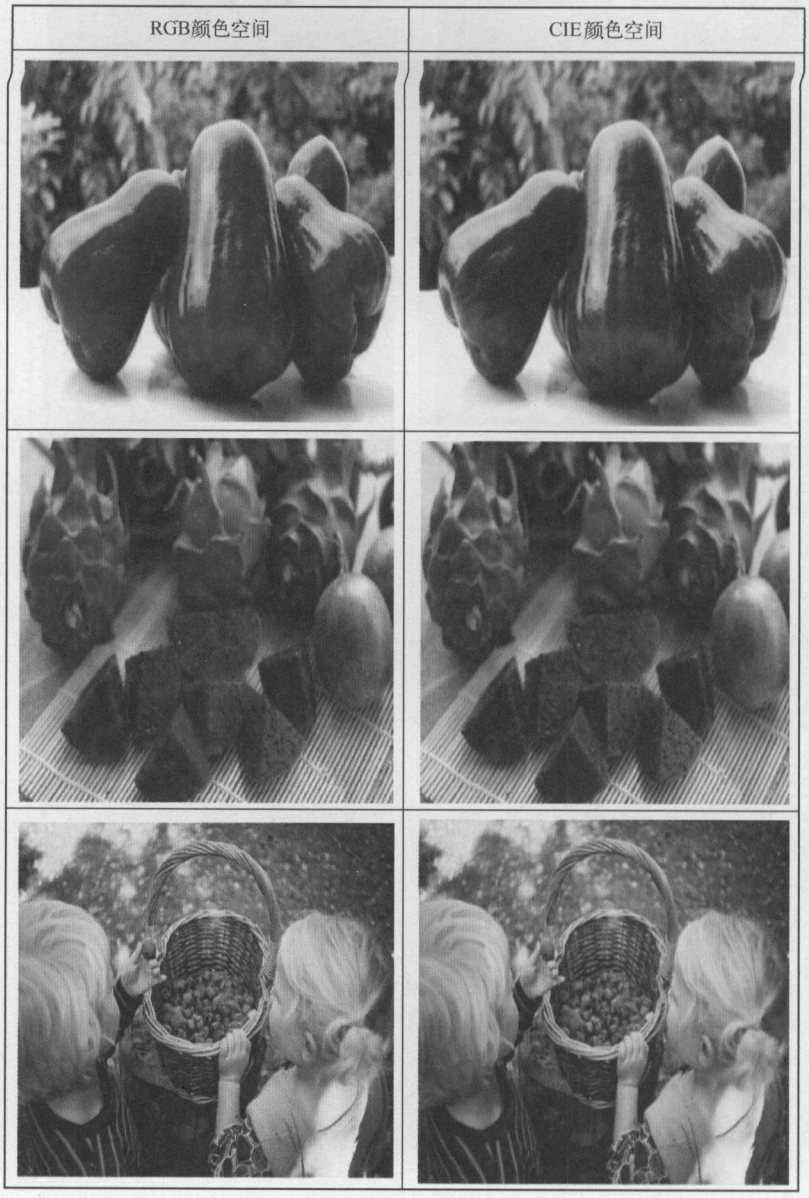


图 10-7　RGB 和 CIE 颜色空间的效果

10.4　图像混合

1. 介绍

图像混合是图像处理技术中的一个技术名词，不仅用于广泛使用的 Photoshop 中，也应

用于 Illustrator、Dreamweaver、Fireworks 等软件。图像混合就是将两幅图像相结合，结合的方式有很多种，被结合的图像称之为基础图像或底层图像，结合的图像称为混合图像。图像混合的主要功效是可以用不同的方法将混合图像颜色与基础图像的颜色混合，运用不同的混合模式，可以看到不同的混合模式的效果。常用的图像混合模式有正常、背面、清除、变暗、变亮、正片叠底、屏幕、颜色加深、颜色减淡、叠加、柔光、强光、相加、相减、差值、反差值、排除等。

2. 正常模式

图像混合的正常模式经常用作默认的混合模式。在此模式下，最后的结果色和基色的不透明度密切相关。在基色的不透明度是 100%，也就是完全不透明的情况下，在“运算”或“应用图像”时使用该模式完全不加混合地将源图层覆盖到目标图层，也就是用源完全替代目标。在基色存在透明度 $a\%$ 时，混合的运算方式是：结果色 = 基色 * $a\%$ + 混合色 * (1 - $a\%$)。

在 OpenGL 中实现正常模式非常简单，顶点着色器无须做额外的工作，只需要读入顶点的纹理坐标即可，像素着色器根据正常模式的运算公式，计算最终的着色值。

```
gl_FragColor      = baseColor * baseColor. a + blendColor * (1 - baseColor. a);
```

其中，baseColor 为基础图像颜色，blendColor 为混合图像颜色。

3. 背面模式

背面模式只有在基础图像完全透明时才会选择对象图像颜色值。可以将基础图像看作一张干净的醋酸纸，这个模式的效果就像是把混合图像画到醋酸纸的背面，只有在透明的像素后面画的区域才是可见的。

```
gl_FragColor    = (baseColor. a == 1. 0)? blendColor:baseColor;
```

4. 清除模式

清除模式总会使用对象颜色值，并将透明度设为 0。

```
vec3 color      = blendColor. rgb;
gl_FragColor    = vec4(color,0. 0);
```

5. 变暗模式

比较两个值，对各部分选择最小的那个值，这个操作会使图像变得更暗。完全为白色的混合图像不会改变基础图像，任意一个图像中的黑色区域都为导致结果为黑色。这个模式是可交换的，即交换混合图像和基础图像结果一样。

```
gl_FragColor      = min(baseColor,blendColor);
```

6. 变亮模式

可以认为变亮模式和变暗模式是相反的，它不是获取各部分的最小值，而是获取各部分的最大值，这个操作会使图像变得更亮。完全为黑色的混合图像不会改变基础图像，任意一个图像中的白色区域都会导致结果为白色。这个模式也是可交换的，即交换混合图像和基础图像结果一样。

```
gl_FragColor  = max(baseColor,blendColor);
```

7. 正片叠底模式

将两个值相乘，这将会在两个图像不是全白的所有区域产生更暗的结果。因为与白色相乘的任何颜色都将会是原始的颜色，所以白色实际是一个恒等操作数。任意一个图像中的黑色区域都会导致结果为黑色。同样这个操作是可交换的。

```
gl_FragColor = baseColor * blendColor;
```

8. 屏幕模式

屏幕模式用白色减去输入的颜色，然后把得到的结果相乘，从而得到最终的颜色。它会将两个输入值的相反值相乘，然后将这一乘法的结果反转，以便产生最终的结果。因为任意一种颜色乘以黑色的反色（白色）都将是原始的颜色，所以黑色实际上是一个恒等操作数。这个模式也是可以交换的。

```
gl_FragColor    = vec4(white - (white - baseColor) * (white - blendColor));
```

9. 颜色加深模式

颜色加深模式会通过降低亮度，根据混合图像的颜色来加深基础图像的颜色。如果基础图像的值为白色，那么这个模式没有效果。

```
gl_FragColor = white - (white - baseColor)/blendColor;
```

10. 颜色减淡模式

颜色减淡模式会通过提高亮度，根据混合图像的颜色来减淡基础颜色。如果基础图像颜色为黑色，那么这个模式没有效果。

```
gl_FragColor    = baseColor/(white - blendColor);
```

11. 叠加模式

叠加模式首先会计算基础图像的亮度，如何亮度值小于0.5，则将混合图像和基础图像相乘。如果亮度值大于0.5，则执行一个屏幕模式操作。其效果就是将混合图像和基础图像值进行混合而不是取代基础图像。这就使得图案和颜色可以覆盖基础图像，但基础图像中的阴影和高光将会保留。当亮度值为0.5时，会出现不连贯的情况，为了实现平滑变换，实际上会对范围在0.45～0.55中的亮度执行两个方程的线性混合。

```
float luminance = dot(baseColor,lumCoeff);
if(luminance <0.45)
gl_FragColor =2 * baseColor * blendColor;
else if(luminance >0.55)
gl_FragColor  = white -2 * (white - baseColor) * (white - blendColor);
else
{
    vec4 result1 =2 * baseColor * blendColor;
    vec4 result2 = white -2 * (white - baseColor) * (white - blendColor);
    gl_FragColor = mix(result1,result2,(luminance -0.45) * 10);
  }
```

12. 柔光模式

柔光模式产生的效果类似于将一束柔光穿过混合图像并射到基础图像上，并依据混合图

像决定使原始图像变亮还是变暗，混合图像亮则更亮，暗则更暗。得到的图像实际上是两个图像的减弱结合。

```
gl_FragColor    = 2 * baseColor * blendColor + baseColor * baseColor -
                  2 * baseColor * baseColor * blendColor;
```

13. 强光模式

强光模式和叠加模式是完全相同的，只不过是使用混合图像来计算亮度值，而不是用基础图像计算亮度值。其效果类似于将一束强光穿过混合图像并射到基础图像上。混合图像中亮度值为 0.5 的像素对基础图像没有效果，这个模式经常会用来制作浮雕的效果，对范围在 0.45 ~ 0.55 中的亮度执行两个方程的线性混合。

```
float luminance = dot(blendColor,lumCoeff);
if(luminance < 0.45)
gl_FragColor = 2 * baseColor * blendColor;
else if(luminance > 0.55)
gl_FragColor  = white - 2 * (white - baseColor) * (white - blendColor);
else
{
    vec4 result1 = 2 * baseColor * blendColor;
    vec4 result2 = white - 2 * (white - baseColor) * (white - blendColor);
    gl_FragColor = mix(result1,result2,(luminance - 0.45) * 10);
}
```

14. 相加模式

在相加模式中，结果是混合图像和基础图像的和。因为得到的值可能会超过 1，所以可能会出现阶段现象。可以交换混合图像和基础图像，结果是一样的。

```
gl_FragColor      = blendColor + baseColor;
```

15. 相减模式

相减模式会从基础图像中减去混合图像。因为得到的值可能会小于 0，所以可能会出现截断现象。

```
gl_FragColor    = baseColor - blendColor;
```

16. 差值模式

在差值模式中，结果是混合图像值和基础图像值之间的差值的绝对值。结果为黑色意味着两个值相等，结果为白色意味着两个值相反。完全相同的图像将会产生一个完全为黑色的结果。一个全白的混合图像可以用来反转基础图像。与黑色混合不会产生任何变化。这个模式也是可以交换的。

```
gl_FragColor  = abs(baseColor - blendColor);
```

17. 反差值模式

反差值模式与差值模式是相反的操作。黑色和白色的混合值会产生于差值模式相同的结

果，但是位于白色和黑色之间的颜色。

```
gl_FragColor = white - abs( white - baseColor - blendColor) ;
```

10.5 邻域平滑

1. 介绍

图像平滑又称图像去噪，其作用是用来消除图像中的高频成分，因为噪声一般都是高频成分。一幅图像中分为高频和低频部分，高频是指图像的亮度变化幅值大的区域，反映在图像上即是边缘的效果和一些噪声，低频则相反，是图像的背景，也叫平滑区域。变化越尖锐的地方高频频谱越多，图像细节就是变化尖锐的地方。图像平滑能够使图像亮度平缓渐变，减小突变梯度，是改善图像质量的图像处理方法，一般用于去除噪声。

邻域平滑是指某像素值取值为邻域内所有像素值的平均值，邻域像素可以取 3 ×3 或 5 ×5的区域，相应的模板为 3 ×3 或 5 ×5。在使用模板进行图像处理时会涉及模板、卷积、卷积核等概念。模板是图像在空间域处理中经常用到的一种方法，模板就是一个矩阵，其数学含义是一种卷积运算。卷积核是指卷积时使用到的权用一个矩阵表示，该矩阵是一个权矩阵。卷积运算可看作加权求和的过程，使用到的图像区域中的每个像素分别于卷积核（权矩阵）的每个元素对应相乘，所有乘积之和作为区域中心像素的新值。因为领域平滑指某像素值取值为邻域内所有像素值的平均值，所以卷积核内各个元素都为 1，邻域内的每一个像素对中间像素的贡献值都是一样的，这样通过综合邻域内像素值来确定中心像素的值来达到平滑的目的。

例如，一个 3 ×3 的邻域平滑模板为：

1	1	1
1	1	1
1	1	1

该模板与图像卷积后得到的值会除以 9，所以领域平滑的效果就是使用相等的权重对一个区域内的所有像素求平均值。平滑去噪的同时也容易模糊高频成分，所以这种平滑又称图像模糊。

下面是 5 ×5 的邻域平滑模板：

1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1

该模板与图像卷积后得到的值除以 25。

2. 算法分析

实现邻域平滑主要是实现图像与模板的卷积，顶点着色器获得顶点坐标后，在像素着色

器利用模板和图像像素值进行卷积运算。下面是具体的步骤分析。

1）顶点着色器

顶点着色器任务就是计算纹理坐标。

第一步：计算顶点位置。

```
gl_Position        = ftransform();
```

第二步：计算纹理坐标。这里用内置的变量 gl_MultiTexCoord0 获取纹理单元上的纹理坐标并存储在内置易变变量 gl_TexCoord[0]中，是像素着色器获取颜色值的基础。

```
gl_TexCoord[0]                = gl_MultiTexCoord0.st;
```

2）像素着色器

像素着色器要实现邻域平均，根据从顶点着色器计算的纹理坐标获取纹理值，并和邻域平滑模板做卷积，再将结果求平均，即获得最终的颜色值。

第一步：获得纹理值。用内置函数 texture2D 计算纹理值，该函数需要两个参数：一个是纹理句柄；另一个是纹理坐标。根据模板的大小依次求和。

```
for (i = 0; i < KernelSize1; i++)
    sum += texture2D(BaseImage, gl_TexCoord[0].st + Offset1[i]);
```

第二步：计算平滑值。在第一步中我们求得了纹理值与模板的卷积，然后求平均就得到了该点上的颜色值，赋值给内置变量 gl_FragtColor，即最后显示的颜色。

```
gl_FragColor = sum * ScaleFactor1;
```

3. 完整代码

1）顶点着色器

```
varying vec2 TexCoord;
void main()
{
    gl_Position      = ftransform();
    gl_TexCoord[0]   = gl_MultiTexCoord0;
}
```

2）像素着色器

```
const int MaxKernelSize = 25;
uniform vec2 Offset1[MaxKernelSize];
uniform int KernelSize1;
uniform vec4 ScaleFactor1;
uniform vec2 Offset2[MaxKernelSize];
uniform int KernelSize2;
uniform vec4 ScaleFactor2;
uniform sampler2D BaseImage;
uniform float Mode;
void one()//3×3
```

```
{
    int i;
    vec4 sum = vec4(0.0);
    for (i = 0; i < KernelSize1; i++)
    sum += texture2D(BaseImage, gl_TexCoord[0].st + Offset1[i]);
    gl_FragColor = sum * ScaleFactor1;
}
void two()//5×5
{
    int i;
    vec4 sum = vec4(0.0);
    for (i = 0; i < KernelSize2; i++)
    sum += texture2D(BaseImage, gl_TexCoord[0].st + Offset2[i]);
    gl_FragColor = sum * ScaleFactor2;
}
void main()
{
    if(Mode==1)
    one();
    else
    two();
}
```

4. 效果图

如图 10-8 所示为几幅图片采用 GPU 算法进行平滑效果，从中可以看出，原来图片中的噪声经过平滑后都减少了，从而得到了更清晰的图片。

10.6 高斯平滑

1. 介绍

10.5 节中已经知道邻域平滑在去噪的同时模糊了高频成分，即图像边缘会变得模糊，因为邻域内的像素对中心像素的贡献值都一样，所以当处理高频成分图像边缘的时候，图像边缘会变得模糊起来。那么如何解决边缘模糊的情况呢？邻域平滑使用的权重都是相等的，如果改变某些权值是不是就可以达到效果了呢？答案是肯定的，我们将靠近中心的像素值应用较高的权值，远离中心的应用较低的权值就可以达到消除噪声地同时又能很好地保留边缘信息的效果，这样的卷积核就称高斯滤波或高斯平滑。

例如，一个 3×3 的模板为：

1/16	1/8	1/16
1/8	1/4	1/8
1/16	1/8	1/16

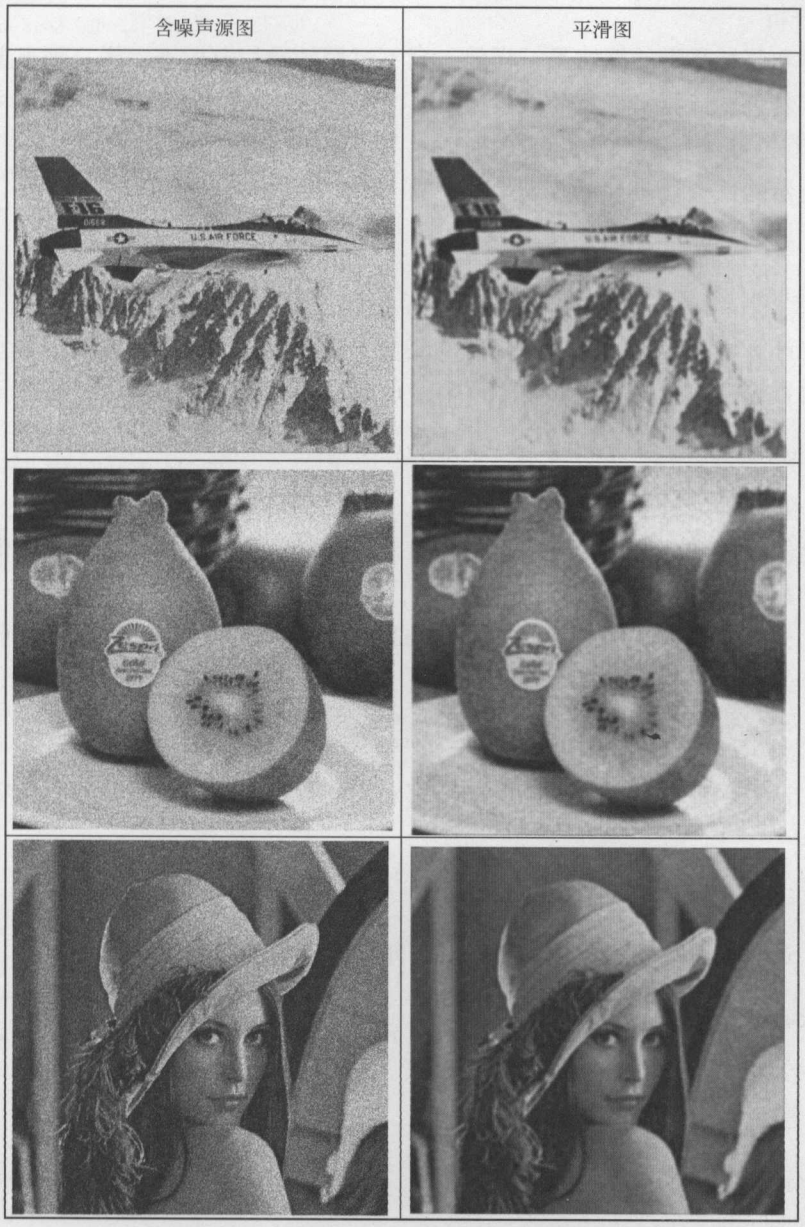


图 10-8　邻域平滑效果

一个 5 ×5 的平滑模板为：

1/273	4/273	7/273	4/273	1/273
4/273	16/273	26/273	16/273	4/273
7/273	26/273	41/273	26/273	7/273
4/273	16/273	26/273	16/273	4/273
1/273	4/273	7/273	4/273	1/273

2. 算法分析

高斯平滑和邻域平滑的算法是一样的，只不过用的卷积模板不一样。顶点着色器读取纹理坐标，像素着色器计算平滑后的像素值。

1）顶点着色器

顶点着色器的任务还是计算纹理坐标。

第一步：计算顶点位置。

```
gl_Position        = ftransform();
```

第二步：计算纹理坐标。这里用内置的变量 gl_MultiTexCoord0 获取纹理单元上的纹理坐标并存储在内置易变变量 gl_TexCoord[0]中，是像素着色器获取颜色值的基础。

```
gl_TexCoord[0]                = gl_MultiTexCoord0.st;
```

2）像素着色器

像素着色器要实现邻域平均，根据从顶点着色器计算的纹理坐标获取纹理值，并和高斯平滑模板做卷积，即获得最终的颜色值。

第一步：获得纹理值。用内置函数 texture2D 计算纹理值，该函数需要两个参数：一个是纹理句柄；另一个是纹理坐标。

```
vec4 tmp = texture2D(BaseImage, gl_TexCoord[0].st + Offset1[i]);
```

第二步：计算平滑值。将得到的纹理值再和高斯模板对应的权值做积，即得到最后的平滑值。

```
sum += tmp * KernelValue1[i];
```

第三步：计算内置变量 gl_FragtColor 值，即最后显示的颜色。

```
gl_FragColor = sum;
```

3. 完整代码

1）顶点着色器

```
varying vec2 TexCoord;
void main()
{
    gl_Position      = ftransform();
    gl_TexCoord[0]   = gl_MultiTexCoord0;
}
```

2）像素着色器

```
const int MaxKernelSize = 25;
uniform vec2 Offset1[MaxKernelSize];
uniform int KernelSize1;
uniform vec4 KernelValue1[MaxKernelSize];
```

```
uniform vec2 Offset2[MaxKernelSize];
uniform int KernelSize2;
uniform vec4 KernelValue2[MaxKernelSize];
uniform sampler2D BaseImage;
uniform float Mode;
void one()//3×3
{
    int i;
    vec4 sum = vec4(0.0);
    for (i = 0; i < KernelSize1; i++)
    {
    vec4 tmp = texture2D(BaseImage, gl_TexCoord[0].st + Offset1[i]);
    sum += tmp * KernelValue1[i];
    }
    gl_FragColor = sum;
}

void two()//5×5
{
  int i;
    vec4 sum = vec4(0.0);
    for (i = 0; i < KernelSize2; i++)
    {
    vec4 tmp = texture2D(BaseImage, gl_TexCoord[0].st + Offset2[i]);
    sum += tmp * KernelValue2[i];
    }
    gl_FragColor = sum;
}
void main()
{
     if(Mode==1)
     one();
     else
     two();
}
```

4. 效果图

如图 10-9 所示是几幅图片经过高斯平滑后的效果，从中可以看出，原来图片中的噪声在高斯平滑之后，减少很多，从而得到的图片具有更少的噪声、更清晰的效果。

10.7　边缘检测

1. 介绍

边缘可以理解为图像中灰度发生突变或不连续的微小区域（一组相连的像素集合），即

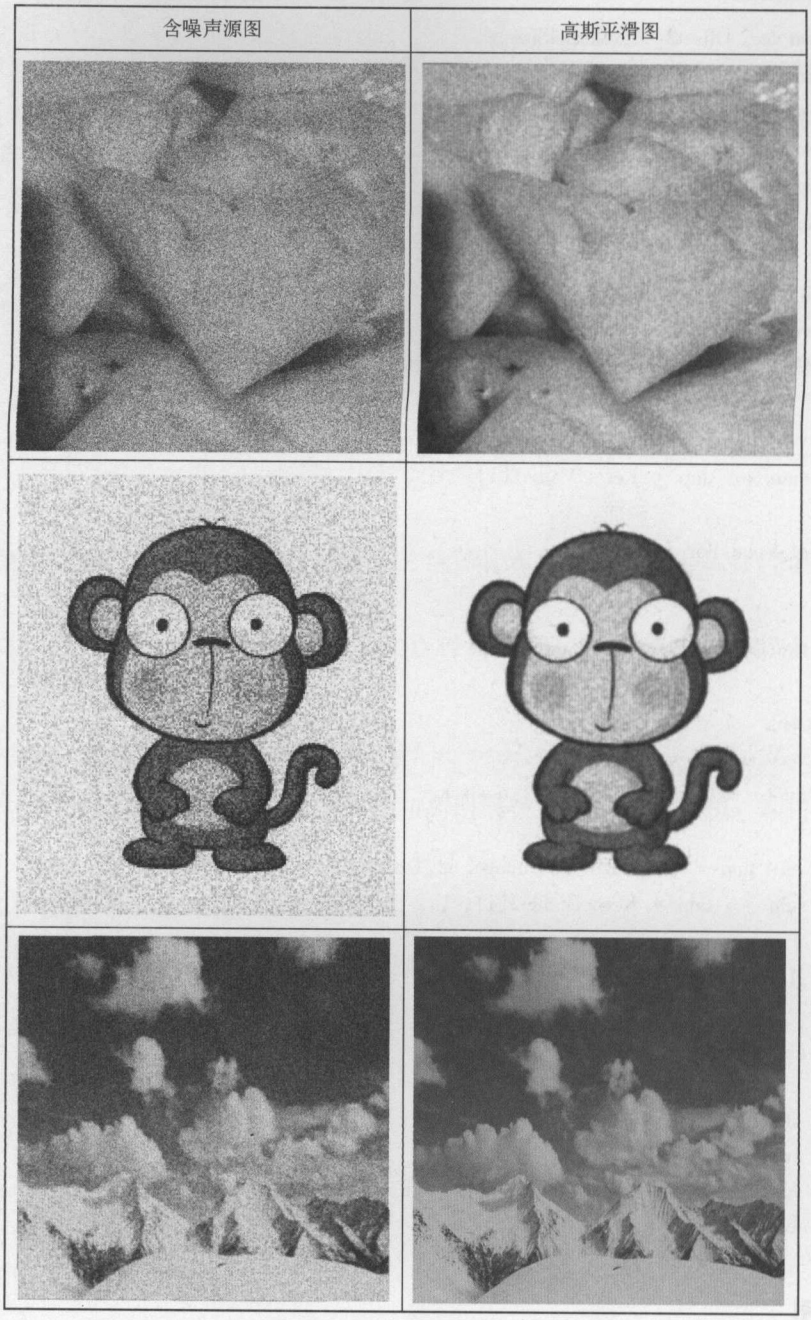


图 10-9　高斯平滑效果

是两个具有相对不同灰度值特性的区域的边界线。边缘是图像的基本特征，它存在于目标与背景、目标与目标之间。边缘检测就是只留住出图像的高频成分，图像的低频成分将会被去除，也就是说，经过边缘检测后图像只留下了“边框”，这也是理解图像的另一种方式。边缘检测同样是用模板与图像进行卷积得到的。

例如，3×3 Laplacian 模板，如下：

1	1	1
1	-8	1
1	1	1

5 × 5 的高斯 - 拉普拉斯模板：

1/8	1/4	1/8	1/4	1/8
1/4	1	2	1	1/4
1/8	2	-15	2	1/8
1/4	1	2	1	1/4
1/8	1/4	1/8	1/4	1/8

2. 算法分析

回顾一下频域平滑和高斯平滑，我们用不同的卷积模板实现了不同的效果，同样，边缘检测也是运用相同的原理，只不过我们用的模板不同而已。

3. 完整代码

1）顶点着色器

```
varying vec2 TexCoord;
void main()
{
    gl_Position     = ftransform();
    gl_TexCoord[0]  = gl_MultiTexCoord0;
}
```

2）像素着色器

```
const int MaxKernelSize = 25;
uniform vec2 Offset1[MaxKernelSize];
uniform int KernelSize1;
uniform vec4 KernelValue1[MaxKernelSize];
uniform vec2 Offset2[MaxKernelSize];
uniform int KernelSize2;
uniform vec4 KernelValue2[MaxKernelSize];
uniform sampler2D BaseImage;
uniform float Mode;
void one()//3 × 3
{
    int i;
    vec4 sum = vec4(1.0);
```

```
        for (i = 0; i < KernelSize1; i++)
        {
        vec4 tmp = texture2D(BaseImage, gl_TexCoord[0].st + Offset1[i]);
        sum -= tmp * KernelValue1[i];
        }
        gl_FragColor = sum;
    }
    void two()//5×5
    {
        int i;
        vec4 sum = vec4(1.0);
        for (i = 0; i < KernelSize2; i++)
        {
        vec4 tmp = texture2D(BaseImage,gl_TexCoord[0].st + Offset2[i]);
        sum -= tmp * KernelValue2[i];
        }
        gl_FragColor = sum;
    }
    void main()
    {
       if(Mode==1)
         one();
       else
         two();
    }
```

4. 效果图

如图 10-10 所示是几幅图片经过 GPU 边缘检测算法后的效果，从中可以看出，GPU 边缘检测算法能够在各种不同内容的图片中很好地检测出图片的边缘。

10.8 锐化

1. 介绍

在图像增强过程中，通常利用各类图像平滑算法消除噪声，一般来说图像的能量主要集中在其低频部分，噪声所在的频段主要在高频段，同时图像边缘信息也主要集中在其高频部分。这将导致原始图像在平滑处理之后，图像边缘和图像轮廓模糊的情况出现。为了减少这类不利效果的影响，就需要利用图像锐化技术，使图像的边缘变得清晰。图像锐化处理的目的是为了使图像的边缘、轮廓线以及图像的细节变得清晰，通常采用的方法是将边缘检测的结果添加到原始图像上，这样会使原始图像的高频成分看起来更加清晰。

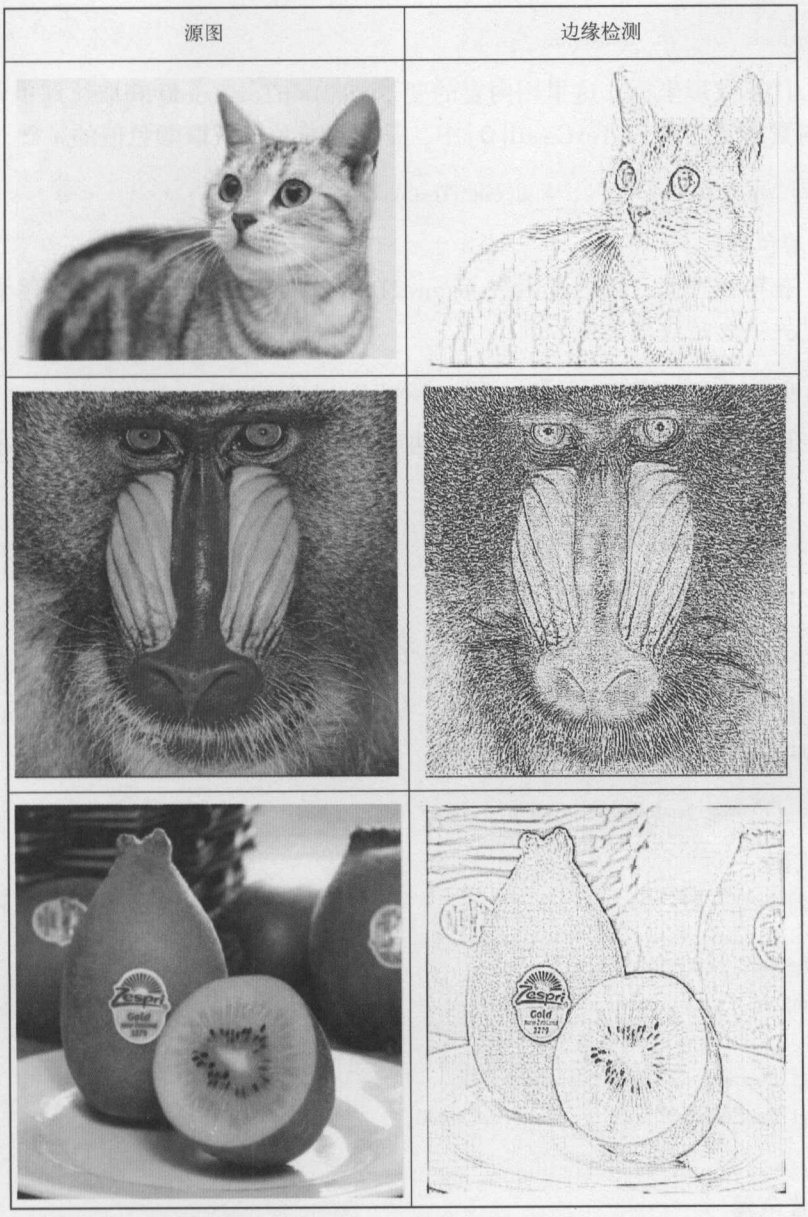


图 10-10　边缘检测效果

2. 算法分析

锐化的核心就是突出边缘信息，那么将边缘检测的结果叠加到源图后就可以实现突出边缘的目的。边缘检测已经在 10.7 节中讲过，在源图上添加边缘检测的结果时，我们可以使用一个缩放因子缩放边缘图像，以此来控制锐化的程度，当要提高锐化的强度时可以增大因子，当减弱锐化强度时可以缩小锐化因子。顶点着色器只需要获取纹理坐标，像素着色器实现边缘检测后再加上原像素值即可实现锐化。

1）顶点着色器

第一步：计算顶点位置。

```
gl_Position        = ftransform();
```

第二步：计算纹理坐标。这里用内置的变量 gl_MultiTexCoord0 获取纹理单元上的纹理坐标并存储在内置易变变量 gl_TexCoord[0]中，是像素着色器获取颜色值的基础。

```
gl_TexCoord[0]              = gl_MultiTexCoord0.st;
```

2）像素着色器

第一步：获得纹理值。用内置函数 texture2D 计算纹理值，该函数需要两个参数：一个是纹理句柄；另一个是纹理坐标。

```
vec4 tmp = texture2D(BaseImage, gl_TexCoord[0].st + Offset1[i]);
```

第二步：计算平滑值。将得到的纹理值再和高斯模板对应的权值做积，即得到最后的平滑值。

```
sum += tmp * KernelValue1[i];
```

第三步：计算原始图像纹理值。

```
vec4 baseColor = texture2D(BaseImage, vec2(gl_TexCoord[0]));
```

第三步：计算内置变量 gl_FragtColor 值，即最后显示的颜色。gl_FragtColor 值是原始图像纹理值和边缘检测图像纹理值缩放后的和。

```
gl_FragColor = ScaleFactor * sum + baseColor;
```

3. 完整代码

1）顶点着色器

```
varying vec2 TexCoord;
void main()
{
    gl_Position      = ftransform();
    gl_TexCoord[0]   = gl_MultiTexCoord0;
}
```

2）像素着色器

```
const int MaxKernelSize = 25;
uniform vec2 Offset1[MaxKernelSize];
uniform int KernelSize1;
uniform vec4 KernelValue1[MaxKernelSize];
uniform vec2 Offset2[MaxKernelSize];
uniform int KernelSize2;

uniform vec4 KernelValue2[MaxKernelSize];
uniform float ScaleFactor;
uniform sampler2D BaseImage;
```

```
uniform float Mode;
vec4 one( )
{
    int i;
    vec4 sum = vec4(0.0);
    for (i = 0; i < KernelSize1; i++)
    {
        vec4 tmp = texture2D(BaseImage, gl_TexCoord[0].st + Offset1[i]);
        sum += tmp * KernelValue1[i];
    }
return sum;

}
vec4 two( )
{
     int i;
    vec4 sum = vec4(0.0);
    for (i = 0; i < KernelSize2; i++)
    {
        vec4 tmp = texture2D(BaseImage, gl_TexCoord[0].st + Offset2[i]);
        sum += tmp * KernelValue2[i];
    }
return sum;
}
void main( )
{
     vec4 baseColor = texture2D(BaseImage, vec2(gl_TexCoord[0]));
    if(Mode == 1)
    {
     vec4 rsum = one( );
      gl_FragColor = ScaleFactor * rsum + baseColor;
    }
     else
     {
      vec4 rsum = two( );
      gl_FragColor = ScaleFactor * rsum + baseColor;
      }
}
```

4. 效果图

如图 10-11 所示是各种不同内容的图片经过 GPU 锐化算法后的效果图，从中可以看

出，图片内容经过锐化之后可以更加清晰，如猫图片里的毛发经过锐化后能够感觉更真实。

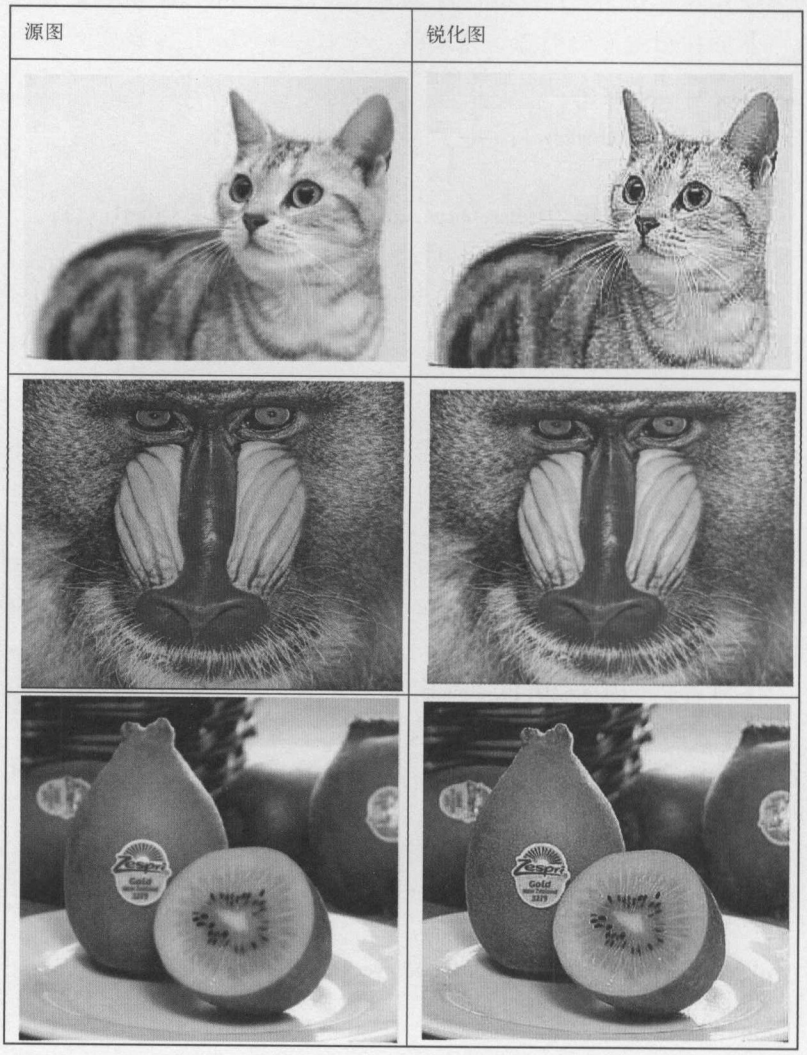


图 10-11　锐化效果

参考文献

[1] OpenGL Shading Language (3rd Edition). July 30, 2009. Randi J. Rost., Bill M. Licea - Kane., Dan Ginsburg, John M. Kessenich.

[2] OpenGL 4 Shading Language Cookbook (Second Edition). December 24, 2013. David Wolff.

[3] OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3 (8th Edition). March 30, 2013. Dave Shreiner, Graham Sellers, John M. Kessenich, Bill M. Licea - Kane.

▶ 作者邮箱：graphicsresearch@qq.com

VR 三维技术系列

VR三维技术系列图书简介

着重底层核心技术的讲解，涵盖三维图形学算法的三大方面，即建模、动画和渲染。读者通过学习本系列专业基础图书和现有成熟计算机基础编程教材，以及三维软件使用的图书，知识体系可以完整地覆盖数字媒体技术专业的所有课程。

书中代码采用C#编程语言，但是本套系列图书里讲述的原理和算法不仅限于C#语言。读者可以通过示例中的代码，采用自己熟悉的编程语言进行编程。本套系列图书包含了很多计算机图形学会议Siggraph论文里最新的、核心的、关键突破和进展的图形学算法讲解、实现与分析。

读者对象

虚拟现实从业人员、数字媒体专业师生、三维游戏工程师、计算机专业师生、软件工程专业师生、影视特效工程师等。

电子信息出版分社微博
http://weibo.com/etpublish

策划编辑：张　迪
责任编辑：底　波
封面设计：徐海燕

官方微信平台

ISBN 978-7-121-31683-8

定价：59.00 元